

AD-A065 087

HONEYWELL INFORMATION SYSTEMS INC MINNEAPOLIS MINN
VIRTUAL MACHINE MONITOR PERFORMANCE ANALYSIS.(U)
DEC 78 S C VESTAL , T KROCAK, H S SCHWENK

F/G 9/2

UNCLASSIFIED

RADC-TR-78-251

F30602-77-C-0097

NL

1 OF 2
ADA
065087



DDC FILE COPY:

ADA065087

Edward A. Lundy
EDWARD A. LUNDY
Project Engineer

Marshall C. Bauman
MARSHALL C. BAUMAN, CPL, USAF
Chief, Information Sciences Division

FOR THE COMMANDER:

John P. Russ
JOHN P. RUSS
Acting Chief, Plans Section

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

19 REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM	
1. REPORT NUMBER RADC-TR-78-251	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER	
4. TITLE (and Subtitle) VIRTUAL MACHINE MONITOR PERFORMANCE ANALYSIS.	5. TYPE OF REPORT & PERIOD COVERED Final Technical Report June 1977 - September 1978		
6. AUTHOR(s) S. C. Vestal, H. S. Schwenk T. Krocak, A. Levy	7. PERFORMING ORG. REPORT NUMBER N/A		
8. PERFORMING ORGANIZATION NAME AND ADDRESS Honeywell Information Systems 2600 Ridgway Parkway Minneapolis MN 55413	9. CONTRACT OR GRANT NUMBER(s) F30602-77-C-0097 NEW		
10. CONTROLLING OFFICE NAME AND ADDRESS Rome Air Development Center (ISIE) Griffiss AFB NY 13441	11. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS 62702F 55810296 1702		
12. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office) Same	13. REPORT DATE December 1978 12 178p.		
14. DISTRIBUTION STATEMENT (of this Report) Approved for public release; distribution unlimited	15. NUMBER OF PAGES 179		
16. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report) Same	17. SECURITY CLASS. (of this report) UNCLASSIFIED		
18. SUPPLEMENTARY NOTES RADC Project Engineer: Raymond A. Liuzzi (ISIE)			
19. KEY WORDS (Continue on reverse side if necessary and identify by block number) Virtual Machines Performance Modeling MULTICS Computers GCOS Operating Systems H6180/6000 Software Engineering Tools			
20. ABSTRACT (Continue on reverse side if necessary and identify by block number) This report describes the H6180 Virtual Machine Monitor Performance Analysis. Included as part of this report is a description of the Virtual Machine Monitor. This report also includes an approach for enhancing the base-line VMM functionality by use of a service machine to control peripheral sharing. The actual experimentation performed in this effort identifies the feasibility of a VMM in a Programming Environment and the performance tradeoffs required for its optimized utilization.			

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

409 346

79 02 23 055

JOB

CONTENTS

Section	Page
I INTRODUCTION	1
Goals and History	1
Summary of Results	3
Personnel	4
II PERFORMANCE ANALYSIS TECHNIQUES	6
Configuration	6
Hardware Modifications for VMM	7
CPU Modifications	7
IOM and Datanet 355 Modifications	13
GCOS Monitoring Tools	14
GCOS Work Load	16
Multics Monitoring Tools	17
Multics Work Load	18
Absentee Processing	18
Job Sequence	19
III ANALYSIS RESULTS	22
Benchmark Analysis	22
GCOS-VMM Overhead Regression Analysis	29
BEST/1 _{tm} -GCOS Analysis	34

CONTENTS (continued)

Section	Page
III	
Multics Benchmarks and Overhead Analysis	44
BEST/1 _{tm} -Multics Analysis	47
Summary of Results	48
IV	
EVOLUTION OF THE VMM	54
VMM Applications	54
Virtual Microcomputers and Minicomputers	54
Networking	55
Program Debugging	55
Input/Output Applications	56
Software Extensions: The Service Machine	59
Hardware Extensions	63
Evolution of Honeywell Computer Products	65
External Influences	65
Internal Influences	66
The Role of the VMM	67
V	
RECOMMENDATIONS	69
Future VMM Research	69
Distributed Systems of Virtual Machines	69
Specialized Virtual Machines	69
Incremental System Extension and Integration	70
Conclusions	71

CONTENTS (concluded)

Section	Page
BIBLIOGRAPHY	73
APPENDIX A. JOB SCRIPTS	75
APPENDIX B. MONITORING TOOLS	97
APPENDIX C. VIRTUAL MACHINE MONITOR PERFORMANCE ANALYSIS: DESIGN PLAN	111

ACCESSION for	
NTIS	White Section <input checked="" type="checkbox"/>
DDC	Buff Section <input type="checkbox"/>
UNANNOUNCED	<input type="checkbox"/>
JUSTIFICATION	
BY	
DISTRIBUTION/AVAILABILITY CODES	
Dist.	MAIL and/or SPECIAL
A	

LIST OF ILLUSTRATIONS

Figure		Page
1	Configuration	6
2	Multics Job Flow	21
3	RADC Multics (VMM) Configuration	23
4	Sample BEST/1 _{tm} Internal Model and Principal Results Report	35
5	Effect of VMM vs. Native GCOS on Response Time as a Function of I/O Activity (I/O Contention)	36
6	Ratio of VMM: Native Response Time for GCOS as a Function of I/O Activity (I/O Contention Included)	37
7	Effect of VMM vs. Native GCOS on Response Time as a Function of I/O Activity (I/O Contention Excluded)	38
8	Ratio of VMM: Native Response Times for GCOS as a Function of I/O Activity (I/O Contention Excluded)	39
9	Effect of VMM vs. Native GCOS on Throughput as a Function of I/O Activity (I/O Contention Included)	40
10	Ratio of VMM: Native Throughput for GCOS as a Function of I/O Activity (I/O Contention Included)	41
11	Effect of VMM vs. Native GCOS on Throughput as a Function of I/O Activity (I/O Contention Excluded)	42
12	Ratio of VMM: Native Throughput as a Function of I/O Activity (I/O Contention Excluded)	43
13	Ratio of VMM: Native Response Time for Multics as a Function of I/O Activity (I/O Contention Included)	49

LIST OF ILLUSTRATIONS (concluded)

Figure		Page
14	Ratio of VMM: Native Response Time for Multics as a Function of I/O Activity (I/O Contention Excluded)	50
15	Ratio of VMM: Native Throughput for Multics as a Function of I/O Activity (I/O Contention Included)	51
16	Ratio of VMM: Native Throughput for Multics as a Function of I/O Activity (I/O Contention Excluded)	52
17	Service Machine Memory Layout	61

LIST OF TABLES

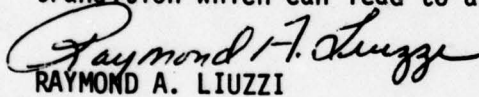
Table		Page
1	Processor and Channel Usages	26
2	Summary of Results for Processor-Bound Work Load	27
3	Summary of Results for Average Work Load	28
4	Summary of Results for Channel-Bound Work Load	29
5	VMM Overhead Regression Data	31
6	Benchmark Data from the PL/1 Load-Control Program	45
7	Estimated VMM Overhead Degradation as a Function of Cache Contribution	46

EVALUATION

This work described in this report and performed during this effort has established that virtual machines can be a significant asset as a software engineering tool in a sophisticated programming environment. This effort, performed on the H6180, has also demonstrated that VMM's in the RADC Programming Environment are feasible with several performance tradeoff factors.

The results from this effort are extremely important in future efforts involving distributed processing as outlined in RADC Technology Plan (TPO V). In a system of logically distributed processing, certain systems in the network will be performing specialized functions on behalf of the other member systems. The role of virtual machines in isolating these functions will be critical to insure optimal performance. As these functions are integrated, virtual machines can again provide the control techniques required to insure compatibility among all systems.

This effort has also demonstrated the role required by virtual machines in hardware/software tradeoffs. Decreasing hardware costs will induce efforts to perform many current software functions in hardware. Virtual machines will play an important role in this transition which can lead to a reduction in software costs.


RAYMOND A. LIUZZI

SECTION I

INTRODUCTION

GOALS AND HISTORY

This document describes The Virtual Machine Monitor Performance Analysis. A Virtual Machine Monitor (VMM) is an operating system which executes on the native hardware and allows other (standard) operating systems to execute in an environment much like the environment an operating system provides for user programs. These sub-operating systems are called virtual machines, or VMs. A VM looks to its contained operating system much like the actual machine looks to the VMM.

While the VMM is aware of the actual hardware complement available, a VM is aware of only the hardware (or simulated hardware in the case of some peripherals) provided to it by the VMM. By providing these virtual environments, a single hardware base can be used by a number of different standard and nonstandard operating systems.

The values of such an arrangement are many, particularly in an environment which engages in research on operating systems. A standard production service can be provided concurrently with an experimental or low-use service. Thus, the need for off-hour scheduling of machine time and interrupted service for boot and re-boot can be reduced.

VMM research in Honeywell and the industry (particularly by IBM: VM/370) has a long history. The Honeywell activity began in the early 1970's as a strategy for product line unification and to provide more flexible internal use of computer equipment.

Interest at Rome Air Development Center (RADC) began in approximately 1974 with an intense look at the security aspects of providing isolated environments for operating systems separated from each other by hardware enforced boundaries. This led to a study of the GCOS Environment Simulator or GCOS encapsulation on Multics. This tool, though not as powerful conceptually as the VMM, is currently in use at RADC.

Business decisions made by Honeywell mandated that the VMM remain in the experimental stages, but low level research continued. In 1976, RADC entered into negotiations with Honeywell's Federal Systems Operations to procure a VMM for further study at RADC. This led to the installation of the VMM at RADC in 1977 and subsequently to the effort described in this document.

Honeywell's Systems and Research Center analyzed the performance of this hardware/software package for RADC to locate the system bottlenecks known to exist and to help RADC plan a strategy for the evolution of VMM research. The findings of that task constitute a detailed analysis of the performance of the Honeywell VMM, some suggestions for improvement should RADC desire to continue the experiment, the results of a model of performance constructed by BGS Systems, Inc., and recommendations for future research.

The remainder of this section contains background and summary information. Section II contains a detailed description of the means used to collect performance data and an analysis of that data. Section III contains plans for the evolution of the VMM and a projection of the role of VMM research in future computer products. The final section details recommendations for extending the VMM capability and continuing research in this area. Appendixes contain the computer listings of the job scripts and monitoring tools used to meter performance, as well as the text of the interim report.

SUMMARY OF RESULTS

Results based on live data experiments and the BEST/1_{tm} analysis show the following:

- VMM overhead has its greatest effect on work loads exhibiting an intermediate amount of I/O activity (15 to 35 connects per second of processor busy time).
- For work loads with a small amount of I/O activity, VMM has only a minor effect on performance. With large amounts of I/O, contention at the I/O devices is the limiting factor.
- In GCOS, the VMM overhead was determined to be in the range of 15 to 28 percent depending upon the I/O mix. This results in an overhead of 3.5 percent of processor busy time and 4.5 msec of overhead per connect.
- In Multics, the VMM overhead was determined to be between 13 and 60 percent. These higher figures are due to the increased dependence of Multics on I/O activity to process page faults.

Based on these figures and the insufficiency of data (particularly in the Multics case), it can be observed that VMM overhead is directly related to I/O activity. Thus, any further work on the performance of this VMM should concentrate on I/O monitoring and speedup.

PERSONNEL

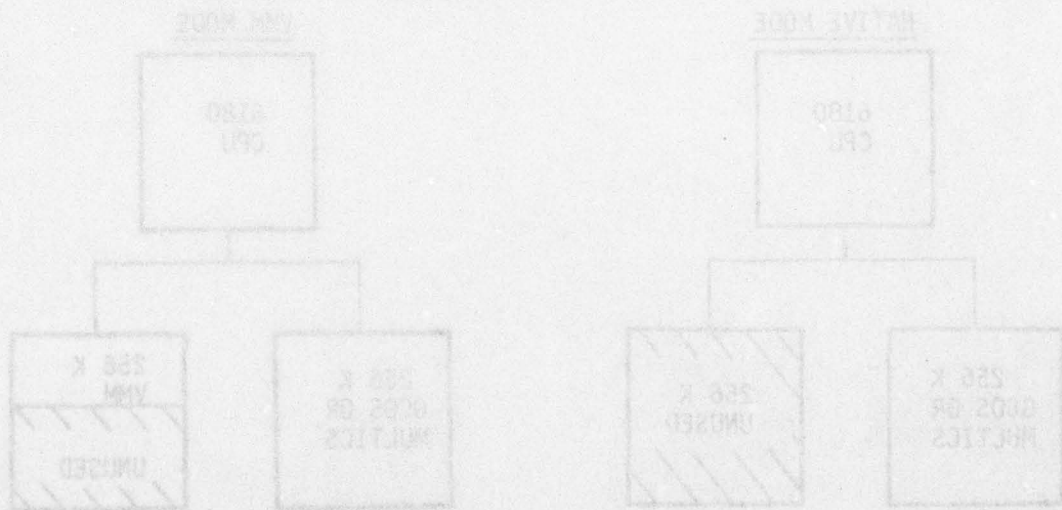
The people involved in this effort were:

- Mr. Ray Liuzzi--contract monitor for this project. He played a valuable role by providing technical direction and leadership in virtual machine research. His interest was instrumental in promoting the potentials of VMMs for Air Force applications.
- Stanley C. Vestal--formerly of the Systems and Research Center, now a scientist with the Corporate Computer Sciences Center. Mr. Vestal served as Principal Investigator for the contract.
- Dr. Harold Schwenk--formerly with Honeywell Information Systems and now the President of BGS Systems, Inc., a subcontractor on this effort. Dr. Schwenk was a co-designer of the VMM.
- Tom Krocak--Honeywell Computer Network Operations. He was responsible for the analysis of GCOS performance.

Additional consulting was obtained from:

- Russ McGee--Honeywell Information Systems. He was the principal designer of the VMM and leader of the VMM program in Honeywell.

- Dr. Robert Goldberg--formerly of Honeywell Information Systems and now with BGS Systems, Inc. Dr. Goldberg is a noted authority in virtual machine concepts and participated in the design of the VMM.
- Larry Shannon--Honeywell Information Systems. Mr. Shannon is an implementor of the VMM.
- Allan Levy--a scientist for BGS Systems, Inc. He was responsible for the BEST/1 modeling and analysis appearing in this document.
- Dr. Jeffrey Buzen--BGS Systems, Inc. Dr. Buzen is an industry leader in techniques of performance analysis and modeling.
- W. Earl Boebert--Systems and Research Center. He assisted with the management of the project for a three-month period.
- Mr. Donald Elefante--RADC. He provided the baseline work load software for Multics.



SECTION II

PERFORMANCE ANALYSIS TECHNIQUES

CONFIGURATION

The RADC Virtual Machine Monitor consists of a modified 6180 processor, IOM, and Datanet 355. The experiments were conducted using a memory configuration of 512 K words and a single processor (Figure 1). When executing a native mode experiment, an operating system was allowed to use 256 K words of memory. During VMM execution analysis, the VMM occupied 256 K and the virtual machine used the second 256 K. In this way the memory utilization for a given operating system was held constant for comparison between native and virtual mode.

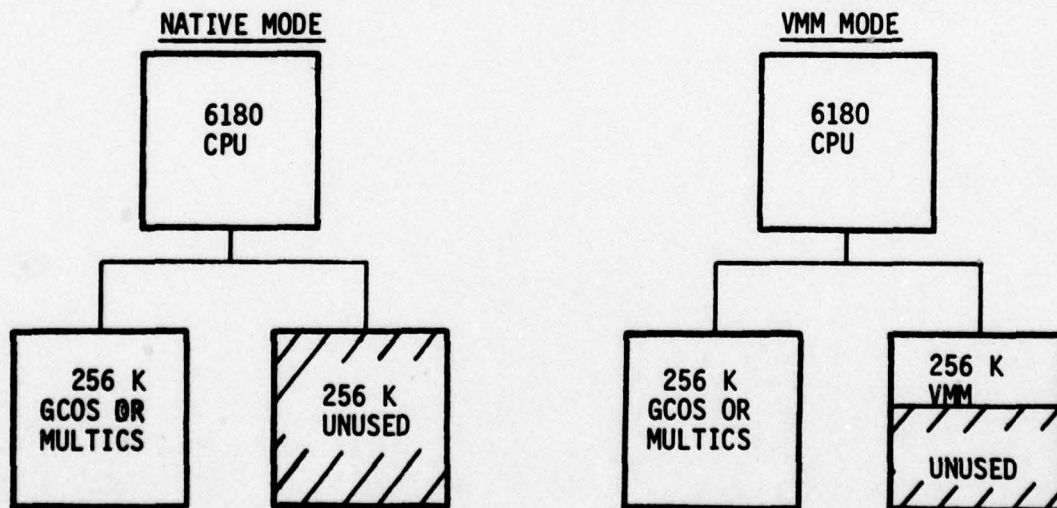


Figure 1. Configuration

Dual operating system performance (GCOS/GCOS, Multics/GCOS, and Multics/Multics) under the VMM was not monitored due to the time constraints of the project.

HARDWARE MODIFICATIONS FOR VMM

The hardware support for VMM exists in the 68/80 CPUs, the IOM direct channels connected to Datanet 355s, and, to a minor extent, in the Datanet 355s. The CPU and Datanet-related changes are described below.

CPU Modifications

The CPU modifications are described under two categories, which is the sequence in which they were implemented.

Category 1 Changes--

Mode Switch Position--The processor mode switch shall have a position added to it. Its positions shall be 6000/6100/VMM (or, if you wish, GCOS/Multics/VMM). The changes described in the following paragraphs shall be active only if the manual mode switch is in the VMM position unless otherwise stated.

Real/Virtual Modes and Indicator--A bit shall be added to the Indicator Register (IR) whose state shall indicate that the processor is in "real" or "virtual" mode. This shall occupy bit 32 of the IR and shall = 1 when in real mode and 0 when in virtual mode. "Real" mode will always be entered upon the execution of a transfer of control instruction while servicing an

interrupt or fault. This will be the only means of entering "real" mode. Once in "real" mode, the execution of an RCU instruction with indicator bit 32 = 0 will be the only instruction which can be used to return to "virtual" mode. Thus this indicator shall not be affected by either the LDI or RET instructions. The state of this indicator will only be storable by the SCU instruction.

Fault on Certain Instructions--Attempted execution of the following instructions in virtual mode shall cause an IPR fault with an illegal in-slave indicator set in the SCU fault conditions:

RSW	SMCM	LCPR	DIS
SSCR	RMCM	SCPR	
RSCR	SMIC	CIOC	

An IPR fault instead of a command fault will occur when any of the above instructions are executed in a GCOS III virtual machine in slave mode.

Category 2 Changes--

VMBAR/VMBND--An additional base/bound register shall be added. The base shall be added to the currently computed final address when executing in virtual mode and the bound shall be used to bound the resulting sum. The register and adder shall be wide enough to apply relocation and addressing of up to 16 million words. The granularity of the VMBAR base and bound shall be 32 K (or preferably less).

LVMBAR/SVMBAR Instructions--These instructions must be added to load and store the VMBAR while in real mode. Their execution in virtual mode shall result in an IPR fault with the illegal in-slave indicator set in the SCU fault conditions.

The LVMBAR will have an op code of 210 and will load the VMBAR from bits 0 through 17 of C(Y). The SVMBAR with an op code of 510 will store the VMBAR in bits 0 through 17 of C(Y) and bits 18 through 35 of C(Y) will be unchanged.

VM Out-of-Bounds Fault--If a virtual machine attempts to address beyond the bound specified by the VMBAR, a store fault with the out-of-bounds indicator set in the SCU fault conditions shall result.

6000/6100 Indicator--Bit 33 of the indicators shall place the processor in either 6000 or 6100 mode. This indicator will be meaningful only when the processor is in VMM or 6100 manual switch position. (The use of this indicator in 6100 switch position is required for the encapsulation of an entire GCOS system under Multics.) The indicator shall = 0 when in 6100 mode and = 1 when in 6000 mode. It will always be stored by the SCU instruction. The SCU instruction is the only instruction which will store the state of this indicator. The indicator shall be loadable only by the RCU instruction, a fault, an interrupt, and the TSS instruction. This indicator shall affect the operation of the processor in the following manner:

- **6100 Mode**--The processor shall execute as a standard 68/80 except as modified as described in this document. This includes the use of indicator bit 28 to indicate the use of the Base Address Register for address development. It shall be the software's responsibility to ensure that this indicator is properly set.
- **6000 Mode**--When in absolute mode, the processor shall operate as a 6000 processor except as modified for the VMM as noted in the previous subsection. Thus, the processor shall perform

addressing relative to the real memory origin or the VMBAR as indicated by the state of the real/virtual indicator.

In append mode the processor shall only be capable of executing valid 6000 instructions. The processor shall perform address development relative to the origin of the segment indicated by the PSR including page relocation if the segment is paged. Addresses shall be further relocated by the base contained in the VMBAR if the processor is in virtual mode.

In addition, the processor shall operate in the following 6000-like manner in spite of other modes and conditions of the processor and specifications in SDWs:

- Instruction bit 29 = 1 shall evoke only the offset part (address register) of pointer registers.
- TRO faults shall occur in slave mode, not in master mode (even if a slave is executing in a privileged segment or a master mode program is executing in an unprivileged segment).
- Instruction bit 28 = 1 shall cause interrupt inhibition (even if executing in an unprivileged segment).
- The BAR shall be loadable in master mode and shall offset effective addresses in slave mode.

Faults and Interrupts--

VMM Switch Position--When a fault or interrupt occurs, the processor will automatically enter Absolute, Real, 6100, or Master Mode during the execution of the vector pair.

However, the corresponding indicators will not be affected unless a transfer of control is executed in the vector pair. If a transfer is executed, the indicators will be modified as described above and as affected by the transfer instruction.

6100 Switch Position--A fault or interrupt when in 6100 position will have the same effect as in VMM position with the exception of the real indicator. This indicator does not exist in this position.

TSS--The TSS instruction shall set the 6000/6100 indicator to 6000 and the master/slave indicator to slave when it is executed in either VMM or Multics manual switch position.

ABSA Instruction--The processor shall execute the ABSA instruction when running in the VMM switch position as follows:

- Add the contents of the VMBAR to the memory address when accessing memory for indirect words, indirect pairs, SDWs, or PTWs. The VMBAR shall be added regardless of the virtual/real indicator state.
- Perform address development according to the settings of the Master/Slave, Absolute/Append, and 6000/6100 indicators when the real/virtual indicator specifies virtual mode.
- Use the Zero, Overflow, and Exponent Underflow indicators to simulate the Master/Slave, Absolute/Append, and 6000/6100 indicators, respectively, when the real/virtual indicator specifies real mode.

- The VMBAR will not be added in the final address development in either virtual or real mode. Thus the address returned by the ABSA shall always be relative to the VMBAR.
- When the real/virtual indicator denotes real mode, the negative indicator will be used to specify the type of address returned by the ABSA instruction. When the negative indicator is off (= 0), the 24-bit absolute operand address will replace the most significant 24 bits of the accumulator, as is normally the case with the ABSA instruction. However, when the negative indicator is set (= 1), an 18-bit effective address will be stored in bits 6 through 23 of the accumulator while zeroes will be placed in the remaining 13 bits. This 18-bit address shall be developed by performing all address development normally associated with the ABSA instruction with the exception of the final append cycle.

During execution of the ABSA instruction, the current state of the processor registers will be used for address development. It is a software responsibility that these be properly loaded before ABSA execution. This implies that the ABSA instruction must be executed in Absolute/Real mode when simulating VM address development within the VMM.

Transfer and Set Virtual Instruction--This instruction when executed in real mode will transfer according to the operand address and enter virtual mode (i. e., bit 32 of the IR shall be reset to 0). When executed in virtual master mode, the instruction will perform as a TRA instruction. Execution of the instruction in virtual slave mode will result in an IPR fault with the illegal in-slave indicator set in the SCU fault conditions. This new

instruction shall have an op code of 715 with bit 27 equal to 1. Currently, the use of this instruction within the VMM is not anticipated; however, it will be used by off-line T&D for the VMM hardware.

IOM and Datanet 355 Modifications

Hardware support is needed to allow a VMM to provide communication support to a virtual machine in such a way that neither the operating system in the virtual machine nor the 355 software need be changed. In addition, this support should prevent the 355 from accessing 6100 memory beyond the bounds of a "window" which is set by the VMM.

The hardware required shall consist of additions to the direct channel which shall modify its operation roughly as follows:

1. When a connect with a mask PCW is delivered to the direct channel from the 6100, it shall retrieve a base and bound from the 6100 memory and then process the mask PCW.
2. Whenever the 355 requests an access to 6100 memory, the direct channel shall relocate and bound check the address to be accessed.

This change accomplishes the desired result for 355s dedicated to virtual machines. The 355, its software, and the operating system continue to operate as before without visibility of the direct channel relocation and bounding. The relocation/bound value is set by the VMM and is unchangeable by either the 355 or the virtual machine it is serving.

Each IOM direct channel which interfaces to a Datanet 355 is modified to store the base and bound addresses of a virtual machine and to assure that all data transfers on behalf of a virtual machine remain within these address limits. In addition, an interface is added between the direct channel and each Datanet 355 which allows the former to deliver an "out-of-bounds" fault to the latter.

GCOS MONITORING TOOLS

Early in the project, it had been assumed that standard GCOS measuring tools could be used to measure the performance of GCOS under VMM. In surveying available tools, one called Peripheral Resource Monitor (PRM) was thought to be the best of those available, particularly in regard to graphical displays. The key measurements centered around processor time both for GCOS and VMM. After some investigation, the "virtual time slippage" problem was investigated in some detail. The essence of the problem is as follows. In GCOS all time is derived from the processor interval timers. These are saved and restored by VMM each time control of the processor is taken from and returned to GCOS. As a result, GCOS maintains accurate virtual processor time, but its real wall clock time will become inaccurate when running under VMM. The effect of "virtual time slippage" on a program like PRM (which measures processor idle directly and processor busy by subtracting idle from elapsed wall time) is that the processor is measured under VMM to be 100 percent busy whether it is 10 percent busy or 100 percent busy.

In an effort to correctly capture GCOS virtual processor time and also provide such VMM data as VMM processor and idle time, a special monitor program was written called VMMON. The key design feature of the program was that VMMON read the system controller clock which is accurate independent of software. This real elapsed time was compared with GCOS time every 15 seconds which under VMM showed a distinct and increasing discrepancy as time continued. This discrepancy was the "virtual time slippage" which is time when GCOS is not in execution, time due to VMM overhead, and/or time when other virtual operating systems are executing. Using this technique, VMMON can accurately measure virtual GCOS processor and idle time and, also, VMM processor overhead in an environment which is relatively processor bound with only one GCOS under VMM. The various tests have proved out this technique for the environment as described.

In an attempt to measure processor utilization for an environment of VMM, GCOS, and Multics, the following steps were taken. Changes to VMM were designed which would record in GCOS memory: VMM processor time, real system idle time, and Multics processor time. VMMON was designed to read this data from GCOS memory and display how this data changed at each sample period. Unfortunately, the required patches to VMM were not able to be debugged given the time constraints of the project.

A copy of the VMMON program is included in Appendix B.

GCOS WORK LOAD

Two fabricated work loads were generated for the GCOS VMM benchmark tests and were designed to produce an average-type load and an I/O bound load. These loads tested the VMM performance under a range of work load types which represent those typical of a data processing site.

In general under GCOS, the amount of system processor overhead (which is not charged to a user program) increases as the level of I/O activity increases. Since VMM intercepts each I/O request and performs some processing for it, the VMM system processor overhead would also increase as the level of I/O activity increases. The fabricated work loads were also designed to be a measurement of this relationship.

The average-type load was composed of four GCOS jobs. All jobs had the same structure with different I/O and processor parameters. Each job was composed of three activities: activity 1 was a compilation of the Fortran driver program with the parameters; activity 2 was an assembly language (GMAP) compilation of a subroutine called by the Fortran driver program which was capable of performing I/O accesses at a maximum rate; activity 3 was the execution of the programs described in activities 1 and 2 which actually produced the desired work load. The I/O to processor ratios produced by the four jobs (by varying the parameters) were as follows:
job AV001 = 0.25:1; job AV002 = 0.50:1; job AV003 = 1:1; job AV004 = 2:1.
The overall I/O-processor ratio for the average load was 0.9:1.

The I/O or channel bound load was also composed of four GCOS jobs. All jobs were identical except for the job control language which varied the disk device to which the I/O accesses were directed. This was an attempt to maximize I/O accesses while minimizing device contention. The structure of the job was identical to that of the average-type load. The parameters in the Fortran driver program produced an I/O-to-processor ratio of 12:1 (which is very I/O bound). The four job names were CH011, CH012, CH021, CH022.

A copy of the fabricated programs described above is included in Appendix A.

MULTICS MONITORING TOOLS

The principal tool used to monitor the performance of Multics was the PL/1 program termination_overseer. Other monitoring commands such as total_time_meters and page_multilevel_meters were used briefly. These last two system commands substantiated the increase in page fault time as shown in the BEST/1 analysis but proved to be inconclusive in the absolute cases.

The termination overseer program uses the real time consumed by the execution of the job scripts to compute a number called throughput. Throughput is defined as the number of iterations of a given job divided by total time consumed for the job. These numbers for each job script are then averaged to give the total experiment throughput under varying loads.

MULTICS WORK LOAD

The means by which the Multics operating system is driven in order to collect performance data is somewhat different than that of GCOS. Since Multics is primarily an interactive system, a simulation of interactive work loads was used in conjunction with the absentee facility of Multics.

The programs used to create the load and in some cases to collect the data were produced by RADC for monitoring the performance of new software releases. However, they have been modified in several cases to fit the needs of the VMM performance analysis.

Absentee Processing

The absentee facility of Multics is a means by which jobs can be scheduled for deferred processing without the need for interaction on the part of the submitter. These jobs retrieve their needed parameters and responses from a file which has been constructed prior to the execution of the job. By judicious anticipation of the execution of a program, responses can be stored in Multics segments which cause a program to behave in the desired manner. The full complement of Multics commands is available to an absentee process. There are no distinctions between interactive processes and these batch-like absentee processes except that input is from segments rather than from a terminal. In fact, absentee processes may be considered to be interactive processes which are not attached to a terminal.

Job Sequence

The programs described here can be found in the appendixes: Job Scripts and Monitoring Tools.

An `exec_com` or command line file (`ec`) called "setup" is first executed to determine the desired load factors which control the experiment. Scheduling parameters at this phase include the start time of the experiment, the number of processes, and the number of interactions desired for each process which will be scheduled. Once this has been determined, the requested number of processes are entered into the absentee queue and scheduled for execution at the desired time. The final function of setup is to call into execution a monitoring program called `termination_overseer`. This monitor will be explained in the section on Multics Monitoring Tools.

The job which has been scheduled for execution is named `load_overseer_n` where `n` is an integer tag indicating the number of the particular process. For example, if three processes are desired to be executed simultaneously, then these absentee jobs queued are named `load_overseer_1`, `load_overseer_2`, and `load_overseer_3`, respectively. The `load_overseer` prototype consists of a second `exec_com` which establishes the proper working directory for the experiment and executes a driver program to control the actual load of a particular process.

The driver program is called `load_control`. This program creates and initializes the necessary files for collecting the data during a single experiment. The clock is read before and after each iteration of an internal loop and the real time used is accumulated in a data segment. The internal

loop is controlled by the iteration count specified as an argument to the setup procedure. At each pass through the loop, a program to exercise the system is called. This exerciser program is called load. Load uses a standard Multics performance monitoring device called flush. Flush modifies each of 256 pages in the user's process space and then compares the results. In this way, at least 256 page faults occur. Additionally, load executes a sequence of processor instructions designed to cause processor-bound activity a variable number of times (set to 100 for these experiments).

When load terminates after causing the paging and processor activity, return is made to the load_control iteration loop. This process is continued until the desired iteration count for this particular absentee job has been exhausted. Load_control then accumulates and computes the throughput factor for this job and terminates. This concludes the life of a single process.

Once all processes have terminated, the termination_overseer computes the total throughput for all absentee jobs and terminates. Figure 2 describes the basic flow through the load experiment.

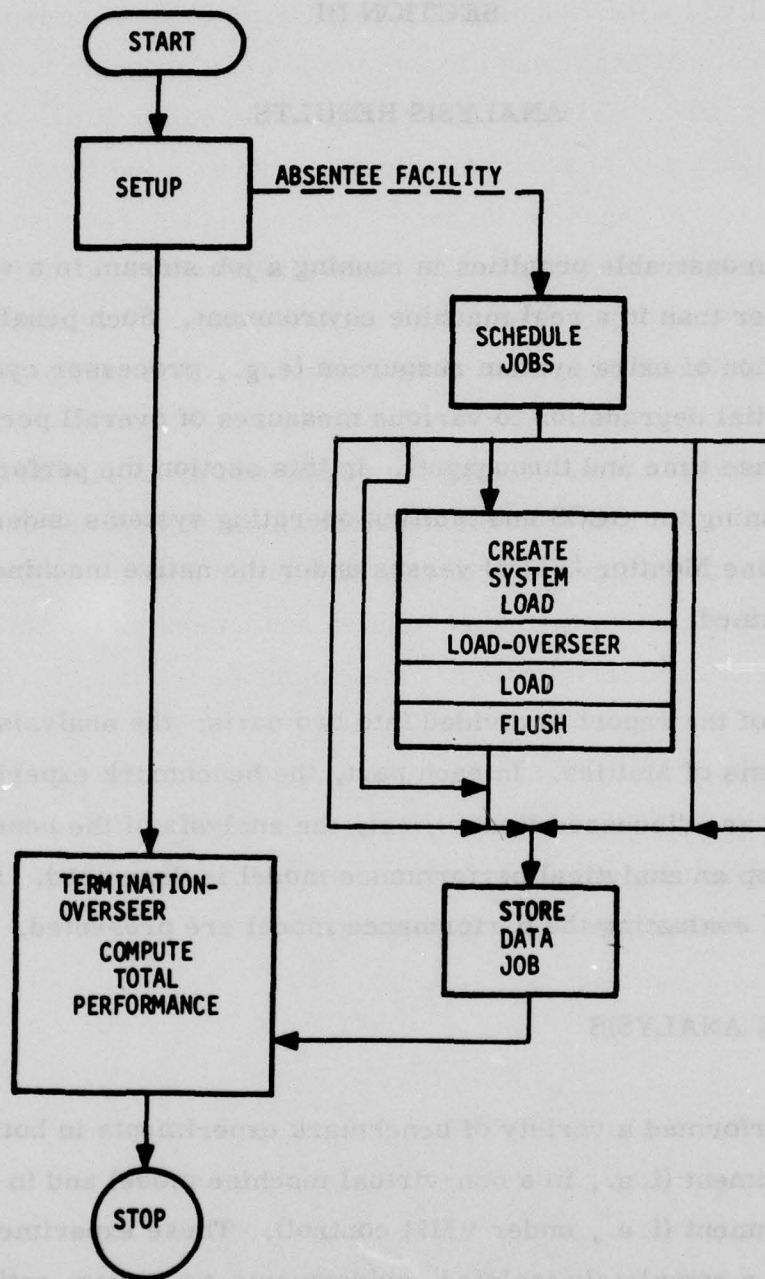


Figure 2. Multics Job Flow

SECTION III

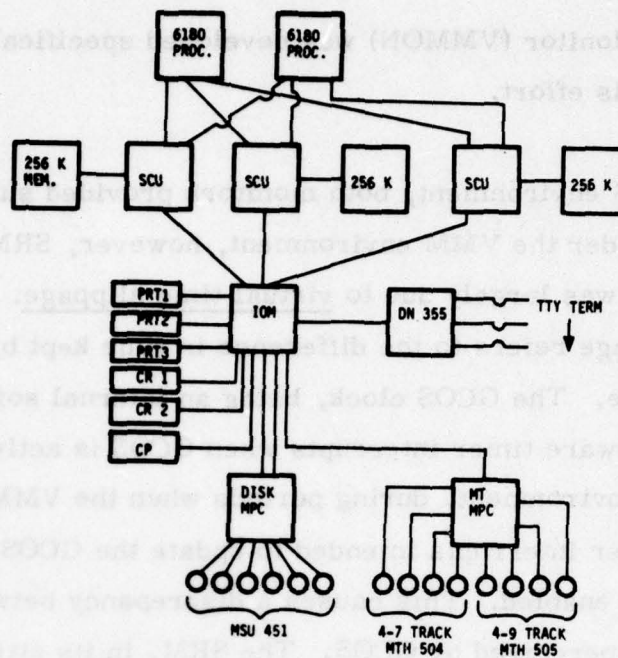
ANALYSIS RESULTS

There are demonstrable penalties in running a job stream in a virtual machine rather than in a real machine environment. Such penalties include the consumption of extra system resources (e.g., processor cycles) as well as potential degradation to various measures of overall performance (e.g., response time and throughput). In this section the performance impact of running the GCOS and Multics operating systems under the Virtual Machine Monitor (VMM) versus under the native machine environment is examined.

This section of the report is divided into two parts: the analysis of GCOS and the analysis of Multics. In each part, the benchmark experiments run by Honeywell are discussed first. Next, the analysis of the benchmark data to develop an analytical performance model is described. Finally, the results of evaluating the performance model are presented.

BENCHMARK ANALYSIS

Honeywell performed a variety of benchmark experiments in both a native GCOS environment (i.e., in a non-virtual machine mode) and in a VMM/GCOS environment (i.e., under VMM control). These experiments were performed in a completely isolated environment; no system activity other than that defined in the benchmark was present. The hardware configuration on which the benchmarks were run is presented in Figure 3.



ONLY A SINGLE PROCESSOR WAS USED DURING THE BENCHMARK.

Figure 3. RADC Multics (VMM) Configuration

Three test series were run on both native GCOS (on a Multics system with VMM processor) and GCOS under VMM. The test series spanned the spectrum of processor bound to channel bound, thus reflecting how VMM performs under a variety of loads. The three separate benchmarks consisted of:

1. Compute-Bound Activity--heavily CPU demanding tasks.
2. Channel-Bound Activity--heavily I/O demanding tasks.
3. Average-Load Activity--average CPU and I/O demanding tasks.

Two software monitors were used during each benchmark experiment. The System Resource Monitor (SRM), an updated version of a previous monitor (CSMON), was available from Honeywell Information Systems. The

Virtual Machine Monitor (VMMON) was developed specifically for the benchmark analysis effort.

In the native GCOS environment, both monitors provided substantially the same results. Under the VMM environment, however, SRM was completely inadequate. This was largely due to virtual time slippage. Basically, virtual time slippage refers to the difference in time kept by GCOS and the actual or real time. The GCOS clock, being an internal software clock, is updated by hardware timer interrupts when GCOS is active. In the virtual machine environment, during periods when the VMM is active and GCOS is idle, timer interrupts intended to update the GCOS clock are lost since they are not enabled. This causes a discrepancy between the actual time and the time perceived by GCOS. The SRM, in its attempts to obtain timing information in the virtual machine environment, unknowingly accesses GCOS's software clock. VMMON on the other hand compensates for the situation by accessing the system controller clock, a microsecond clock which remains accurate under both the native/GCOS and the VMM/GCOS environments. In addition, VMMON reports both the value of the GCOS time and the actual time, thereby permitting the virtual time slippage to be deduced.

In the VMM/GCOS environment, the virtual time slippage is caused only by VMM overhead activity and system idle activity. The virtual time slippage, therefore, provides a convenient mechanism for establishing overhead attributable to the VMM. One need only concentrate on periods in the native system where the processor is known to be 100 percent busy. Virtual time slippage will, in this case, accurately represent VMM overhead only.

Two of the benchmark test series run in the native/GCOS environment--the processor bound and the average load activities--managed to consume 100 percent of the processor. In the channel bound tests, however, the native/GCOS was approximately 66 percent idle and so the VMM overhead could not be deduced without additional data. Attempts to obtain this data through applications of various patches to the VMM were not successful.

The actual monitor reported data for all benchmark experiments is provided in Tables 1 through 4. This data has been reduced for use in the performance analysis presented below. Table 1 presents a summary of processor and channel usage data. Tables 2 through 4 present data extracted from VMMON for six time intervals T1 through T6 in both the native/GCOS and VMM/GCOS environments. The following summarizes the information presented:

GCOS Time: time interval captured by GCOS

Real Time: time interval captured by the system controller clock

GCOS Ovhd: percentage of GCOS overhead time

VMM Ovhd: percentage of VMM overhead time

Idle: percentage of processor idle time

Connects: number of connects per second of GCOS time

TABLE 1. PROCESSOR AND CHANNEL USAGES

Channel-Bound Test Series

	Real Time (hr)*	Processor (hr)	Channel (hr)	% Processor	% Channel
Native GCOS	0.714	0.0270	0.187	3.8	26.2
VMM GCOS	0.746	0.0334	0.070	4.4	9.3

Average Load Test Series

	Real Time (hr)	Processor (hr)	Channel (hr)	% Processor	% Channel
Native GCOS	0.611	0.1476	0.132	24.1	21.6
VMM GCOS	0.916	0.1529	0.123	16.7	13.4

Processor-Bound Test Series

	Real Time (hr)	Processor (hr)	Channel (hr)	% Processor	% Channel
Native GCOS	0.149	0.1357	0.019	91.1	12.8
VMM GCOS	0.217	0.1206**	0.013**	55.5	6.0

*For all job activities

**The VMM GCOS tests were terminated early due to time constraints.

**TABLE 2. SUMMARY OF RESULTS FOR PROCESSOR-BOUND
WORK LOAD**

Native GCOS Environment

Interval Statistic	T1	T2	T3	T4	T5	T6
GCOS Time (sec)	94.4	223.2	349.5	128.8	220.7	443.0
Real Time (sec)	94.4	223.2	349.4	128.8	220.7	442.9
% GCOS Ovhd	2.4	1.8	1.7	1.4	1.6	1.7
% VMM Ovhd	0.0	0.0	0.0	0.0	0.0	0.0
% Idle	0.0	0.0	0.0	0.0	0.0	0.0
Number of Connects/sec	5.3	2.9	2.5	1.1	2.0	2.4

VMM/GCOS Environment

Interval Statistic	T1	T2	T3	T4	T5	T6
GCOS Time (sec)	124.9	190.5	286.0	287.5	415.9	479.0
Real Time (sec)	140.4	208.8	306.5	295.4	427.2	501.9
% GCOS Ovhd	3.2	2.7	2.1	0.9	0.9	1.2
% VMM Ovhd	12.4	9.6	7.2	2.7	2.7	4.8
% Idle	0.0	0.0	0.0	0.0	0.0	0.0
Number of Connects/sec	8.3	6.9	4.8	1.2	1.1	2.1

TABLE 3. SUMMARY OF RESULTS FOR AVERAGE WORK LOAD

Native GCOS Environment

Interval Statistic	T1	T2	T3	T4	T5	T6
GCOS Time (sec)	124.2	216.5	339.6	274.5	216.1	340.1
Real Time (sec)	124.1	216.4	339.4	274.3	216.0	339.9
% GCOS Ovhd	10.2	10.0	10.0	9.5	8.7	9.8
% VMM Ovhd	0.0	0.0	0.0	0.0	0.0	0.0
% Idle	0.16	0.14	0.09	0.09	0.12	0.0
Number of Connects/sec	38.3	38.0	38.0	35.9	32.3	37.4

VMM/GCOS Environment

Interval Statistic	T1	T2	T3	T4	T5	T6
GCOS Time (sec)	154.1	278.4	461.0	186.1	306.9	400.2
Real Time (sec)	184.8	334.1	548.9	223.3	364.1	479.8
% GCOS Ovhd	9.9	9.9	9.3	9.9	9.1	9.1
% VMM Ovhd	19.9	20.0	19.1	20.0	18.6	19.9
% Idle	0.0	0.0	0.0	0.0	0.0	0.0
Number of Connects/sec	36.7	36.8	34.7	36.9	33.7	36.6

TABLE 4. SUMMARY OF RESULTS FOR CHANNEL-BOUND
WORK LOAD*

Native/GCOS Environment

Interval Statistic	T1	T2	T3	T4	T5	T6
GCOS Time (sec)	142.9	264.0	324.0	181.0	271.2	479.2
Real Time (sec)	142.8	263.9	323.9	181.0	271.2	479.2
% GCOS Ovhd	17.4	17.6	17.6	17.7	17.6	17.6
% VMM Ovhd	0.0	0.0	0.0	0.0	0.0	0.0
% Idle	65.8	65.8	65.9	65.9	66.1	66.2
Number of Connects/sec	64.0	64.9	64.9	65.5	65.2	65.0

*Since native GCOS idle is significantly greater than zero (approximately 66 percent), VMM overhead for the channel-bound work load cannot be deduced. Attempts to patch VMMON to present the data on the VMM/GCOS environment were unsuccessful.

GCOS-VMM OVERHEAD REGRESSION ANALYSIS

The quantity of interest for the analyses that follow is the VMM overhead. This quantity represents the processor time consumed by the VMM in servicing users' requests for system resources. The approach to be used to compute VMM overhead is described below.

The various time periods for which measurements are available are denoted by T_i ($i = 1, 2, \dots, n$). The measured VMM overhead time (i. e., the virtual time slippage in the i^{th} time period) is denoted by t_i . Suppose

there are M different types of requests which the VMM must service. If n_{ij} ($j=1, \dots, m$) is the measured quantity of requests of type j during the i^{th} interval and θ_j is the VMM overhead incurred in servicing the type j request, then the total VMM overhead time spent in the i^{th} time interval may be approximated by:

$$t_i = \sum_{j=1}^m \theta_j N_{ij} \quad (1)$$

The coefficients $\theta_1, \theta_2, \dots, \theta_m$ can be determined by the method of least squares so as to minimize the sum of the squares of the residuals, i. e.,

$$\min S(\theta_1, \theta_2, \dots, \theta_m) = \sum_{i=1}^n r_i^2$$

where residual r is defined as

$$r_i = t_i - \sum_{j=1}^m \theta_j N_{ij}$$

If the r_i turn out to be relatively small, then Equation (1) is considered to yield a suitable fit to the data. The goodness of fit criteria incorporated below is the multiple correlation coefficient obtained by comparing the sum of the residual squares to the sum of the squares of the deviation of the measurements from their mean value (\bar{t}), i. e.,

$$R^2 = 1 - \frac{\sum_{i=1}^n r_i^2}{\sum_{i=1}^n (t_i - \bar{t})^2}$$

Based upon the information obtained from the benchmark analyses, two resource quantities were selected as being major contributions to VMM overhead. For the i^{th} time interval, T_i , these were

- n_{i1} --the CPU busy time during T_i
- n_{i2} --the number of channel connects requested during T_i

Equation (1) in this case reduces to

$$t_i = \theta_1 n_{i1} + \theta_2 n_{i2} \quad (2)$$

Using the benchmark experiment data in Tables 1 through 4 (excluding the channel-bound work load for reasons previously explained), the method of least squares was applied to the equations represented in Table 5.

TABLE 5. VMM OVERHEAD REGRESSION DATA

Work Load	T_i	VMM Overhead Time (t_i) in T_i (sec)	GCOS Busy Time (n_{i1}) in T_i (sec)	Total Number of Connects (n_{i2}) in T_i
Processor Bound	1	15.5	124.9	1042.9
	2	18.3	190.5	1316.4
	3	20.5	286.0	1381.4
	4	7.9	287.5	345.0
	5	11.3	415.9	474.1
	6	22.9	479.0	1005.9
Average Load	1	30.7	154.1	5661.6
	2	55.7	278.4	10242.3
	3	87.9	461.0	16010.5
	4	37.2	186.1	6876.4
	5	57.2	306.9	10351.7
	6	79.6	400.2	14647.3

The results of the regression analysis of the data in Table 5 yielded the following:

θ_1	θ_2	R^2
0.0351	0.0045	0.98

These results indicate that the processor overhead incurred through usage of the VMM is approximately 0.0045 seconds overhead for each channel connect and an additional 3.5 percent of non-idle processor time. The extremely high correlation coefficient indicates that these results represented a suitable fit to the data.

It should be pointed out that, although an extremely high correlation coefficient has been achieved, the results may not be statistically precise. The method on which the coefficient values are based depends on the assumption that the coefficients are constant--not variable. This assumption is not valid for the data in Table 5. Clearly different user requests place significantly different demands on the system, for example. The true accuracy of these coefficients, therefore, has not been established. It is possible, however, with additional measurements, to construct more statistically accurate results. This remains a source for future investigations.

Another point is also of interest at this time. Much care was taken in obtaining representative time intervals including the effect of transitory periods of system activity rather than just intervals of pure work load activity. This was done to include the effects of transitory system activity in the analyses of subsequent sections. With careful selection of periods

of measurement, it is possible to distinguish between periods of pure work load activity and periods of combined work load and transitory system activity. It may be of interest to determine the coefficients θ'_1 and θ'_2 in this case. The values of θ'_1 would more accurately isolate the contributions of n_{i1} and n_{i2} to the VMM overhead.

For this purpose the following intervals were chosen as representative of pure work load activity (i. e., little or no system transitory effects were included):

Work Load	VMM Overhead (t_i) (sec)	GCOS Busy Time (n_{i1}) (sec)	Number of Connects (n_{i1})
Processor Bound	6.8	287.3	132.9
Average Load	55.7	278.4	10242.3

Repeating the previously described regression analysis determines:

$$\theta'_1 = 0.0215$$

$$\theta'_2 = 0.0048$$

Application of these new coefficients to intervals which include transitory effects yields a significant difference between the observed and predicted VMM overhead. This effect is due to overhead resulting from factors other than the GCOS busy time (n_{i1}) and the number of connects (n_{i2}), that is,

n_{i3}, n_{i4}, \dots which have not been identified. To determine such factors would require substantial additional data and analyses.

BEST/1_{tm}-GCOS ANALYSIS

Based on the values of θ_1 and θ_2 derived in the previous section, it was possible to hypothesize several configuration and work load alternatives and determine their respective performance degradation when executing GCOS in a virtual machine rather than in a native machine environment. This was done with the help of BGS Systems' proprietary modeling package BEST/1_{tm}. First, the performance of each of the hypothesized alternatives running in a native/GCOS environment was analyzed using BEST/1_{tm}. Next the previously determined coefficients, θ_1 and θ_2 , were used to determine the VMM overhead degradation of the hypothesized work loads in the VMM/GCOS environment. Finally, BEST/1_{tm} was again used to determine the performance impact of executing that same configuration and work load, under GCOS, in the virtual machine environment.

The configuration and work load models constructed for the BEST/1_{tm} analysis consisted of a canonical job stream executing on a single processor system, in one case including and in one case excluding the effects of I/O device contention. Individual tasks in the job stream were assumed to consume approximately 1 second of CPU time and to perform a varying number of I/O operations (0 to 50) at approximately 35 μ sec per connect. The analyses were performed for the job stream under several distinct levels of system load, ranging from a single thread environment to a level of threading equal to 15.

The models chosen and analyzed via BEST/1_{tm} were directed toward determining the VMM overhead impact on two important measures of system performance: response time and system throughput. Sample BEST/1_{tm} modeling details and the results of the analyses are presented in Figures 4 through 12.

```

BEST/1>
LIST

-----WORKLOAD 1-----DESCRIPTORS-----
      LABEL PROCESSOR-BOUND
      BP WORKLOAD TYPE
      0.75 ATTAINED MPL

-----WORKLOAD 2-----DESCRIPTORS-----
      LABEL AVEPAGE-LOAD
      BP WORKLOAD TYPE
      0.75 ATTAINED MPL

-----WORKLOAD 3-----DESCRIPTORS-----
      LABEL CHANNEL-BOUND
      BP WORKLOAD TYPE
      1.50 ATTAINED MPL

SERVER
1 CPU          WK1 1 WK2 2 WK3 3
2 DISK2        1000.0 1000.0 1000.0
3 DISK3         94.5   0.0   0.0
4 DISK4         0.0 1281.0   0.0
                0.0   0.0 2271.5

BEST/1>
30

*** PRINCIPAL RESULTS ***

WORKLOAD      RESPONSE TIME      THROUGHPUT      % CPU
1 PROCESSOR-BOUND 1.93 SEC      1399. PRP HOUR   33.3 %
2 AVEPAGE-LOAD    3.47 SEC      773. PRP HOUR    21.6 %
3 CHANNEL-BOUND   5.35 SEC      1009. PRP HOUR   28.0 %
TOTAL CPU UTILIZATION = 83.5 %

```

Figure 4. Sample BEST/1_{tm} Internal Model and Principal Results Report

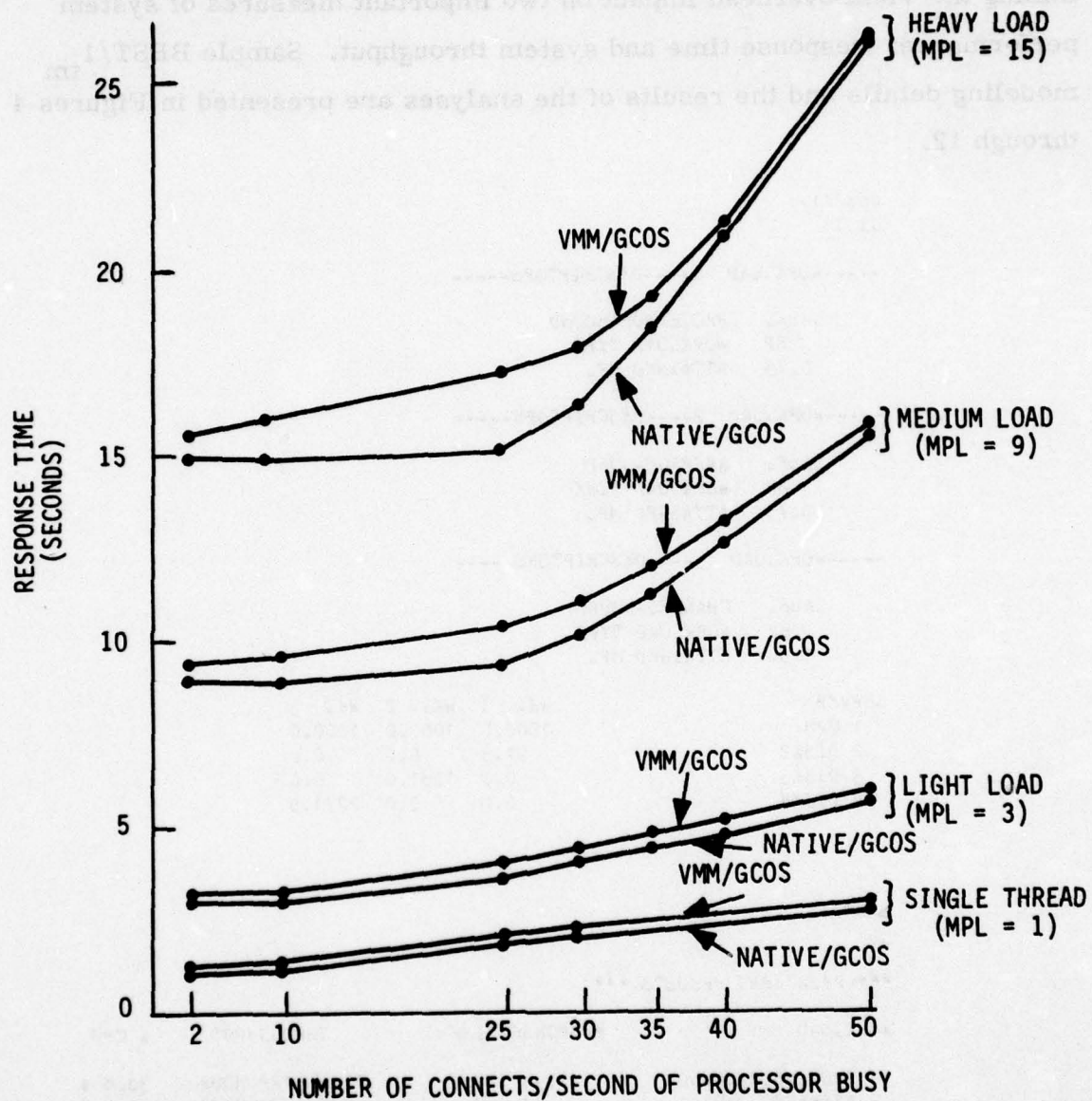


Figure 5. Effect of VMM vs. Native GCOS on Response Time as a Function of I/O Activity (I/O Contention)

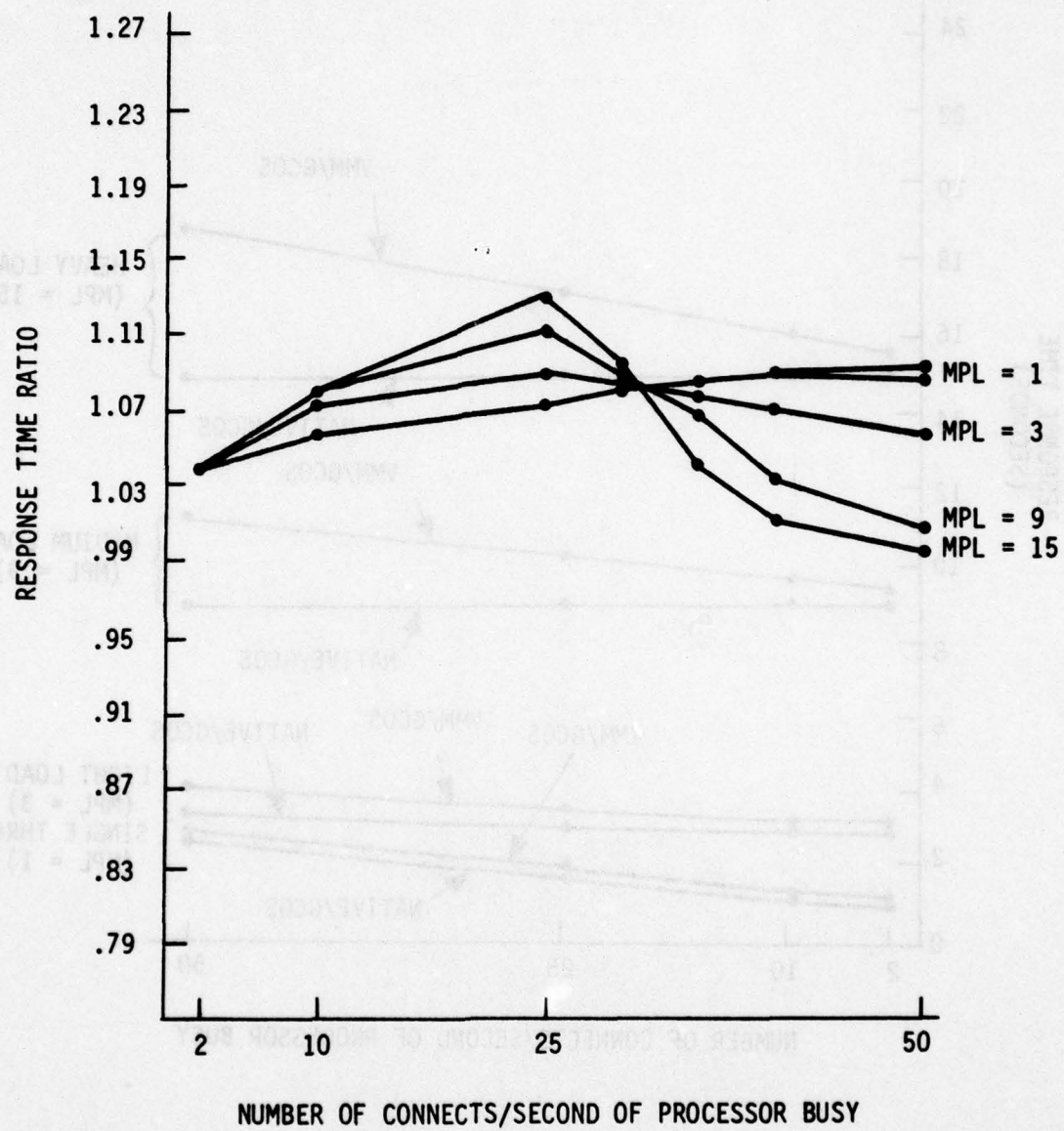


Figure 6. Ratio of VMM: Native Response Time for GCOS as a Function of I/O Activity (I/O Contention Included)

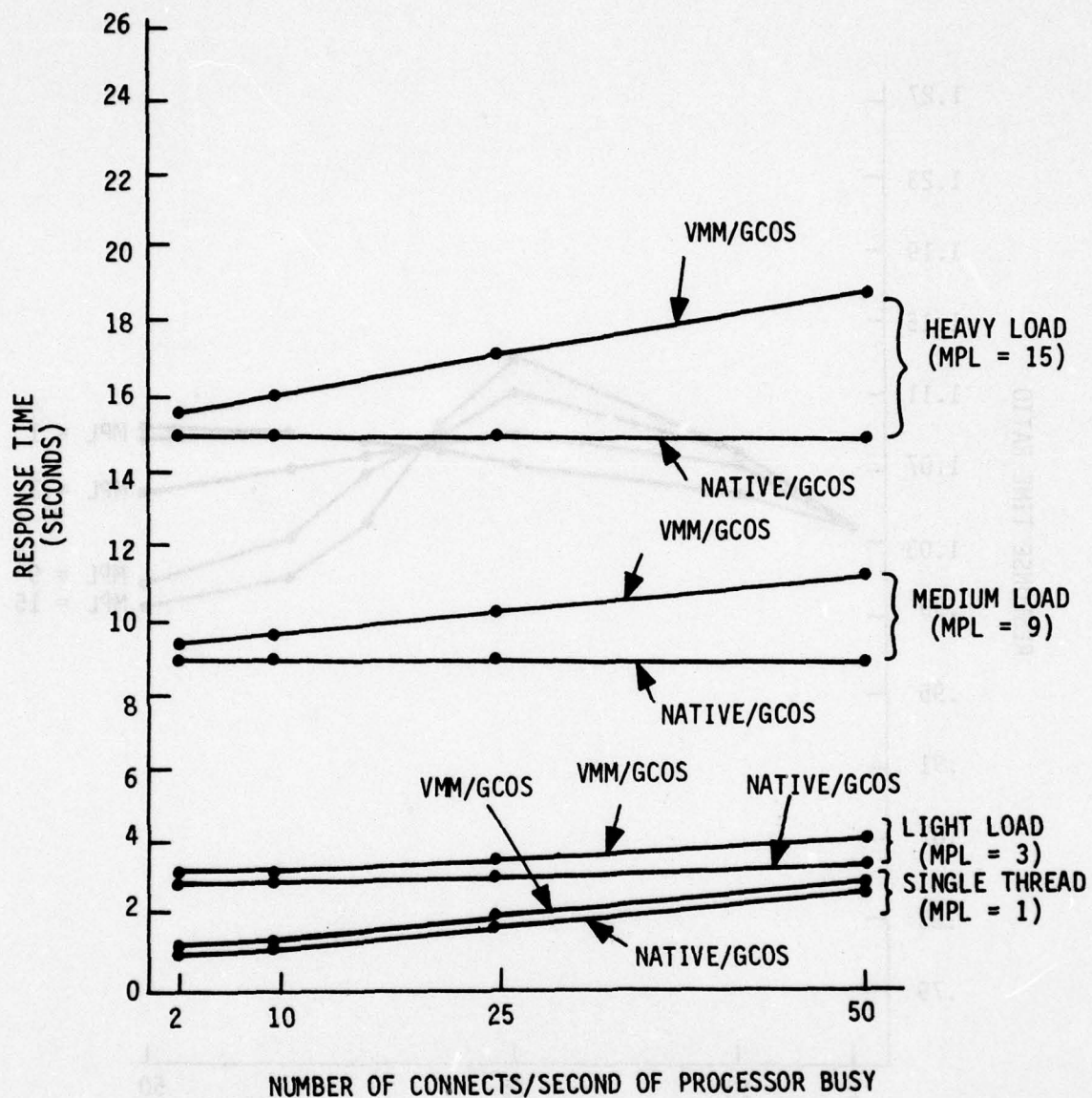


Figure 7. Effect of VMM vs. Native GCOS on Response Time as a Function of I/O Activity (I/O Contention Excluded)

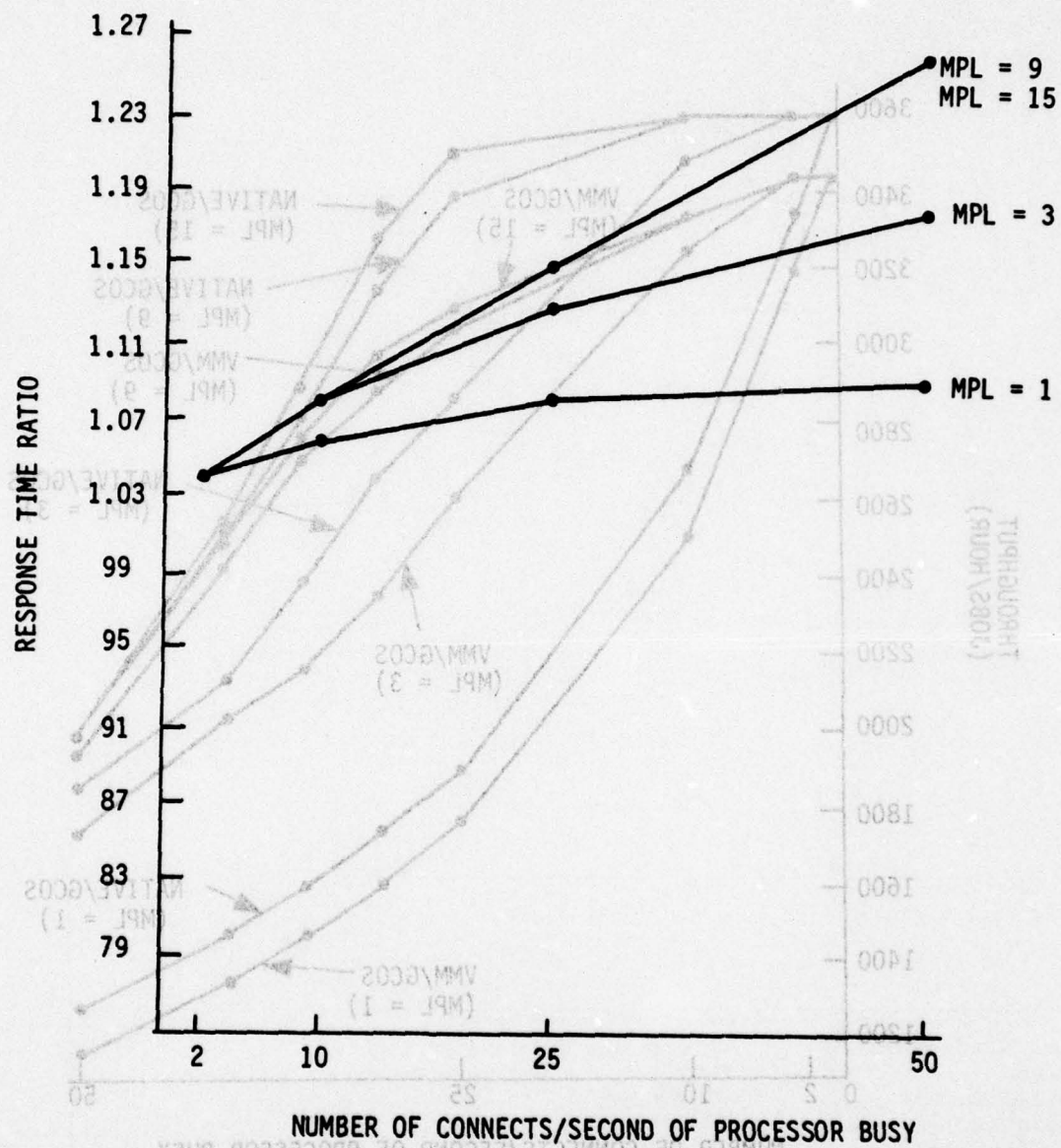


Figure 8. Ratio of VMM: Native Response Times for GCOS as a Function of I/O Activity (I/O Contention Excluded)

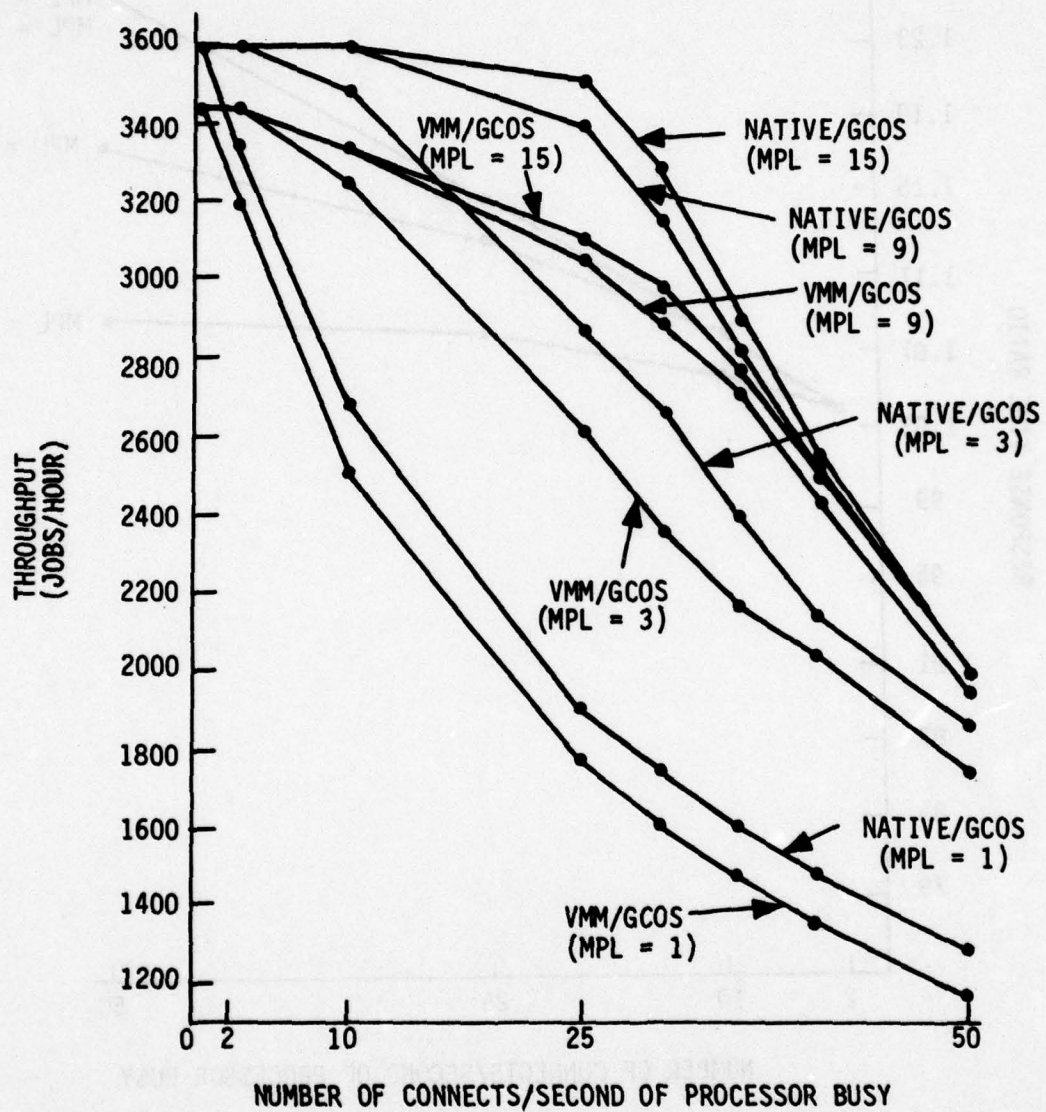


Figure 9. Effect of VMM vs. Native GCOS on Throughput as a Function of I/O Activity (I/O Contention Included)

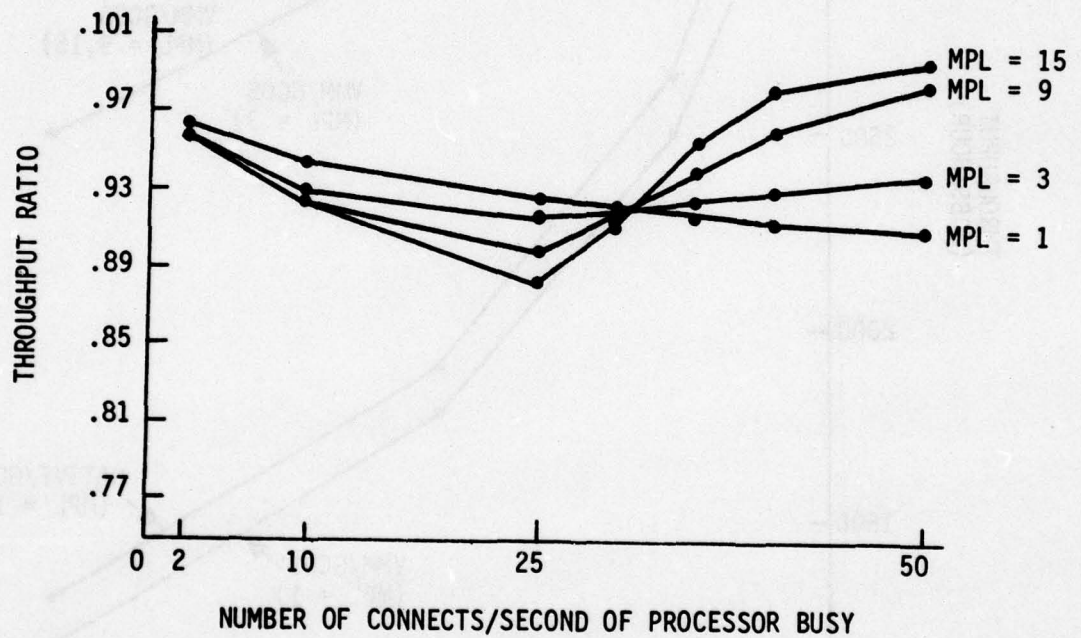


Figure 10. Ratio of VMM: Native Throughput for GCOS as a Function of I/O Activity (I/O Contention Included)

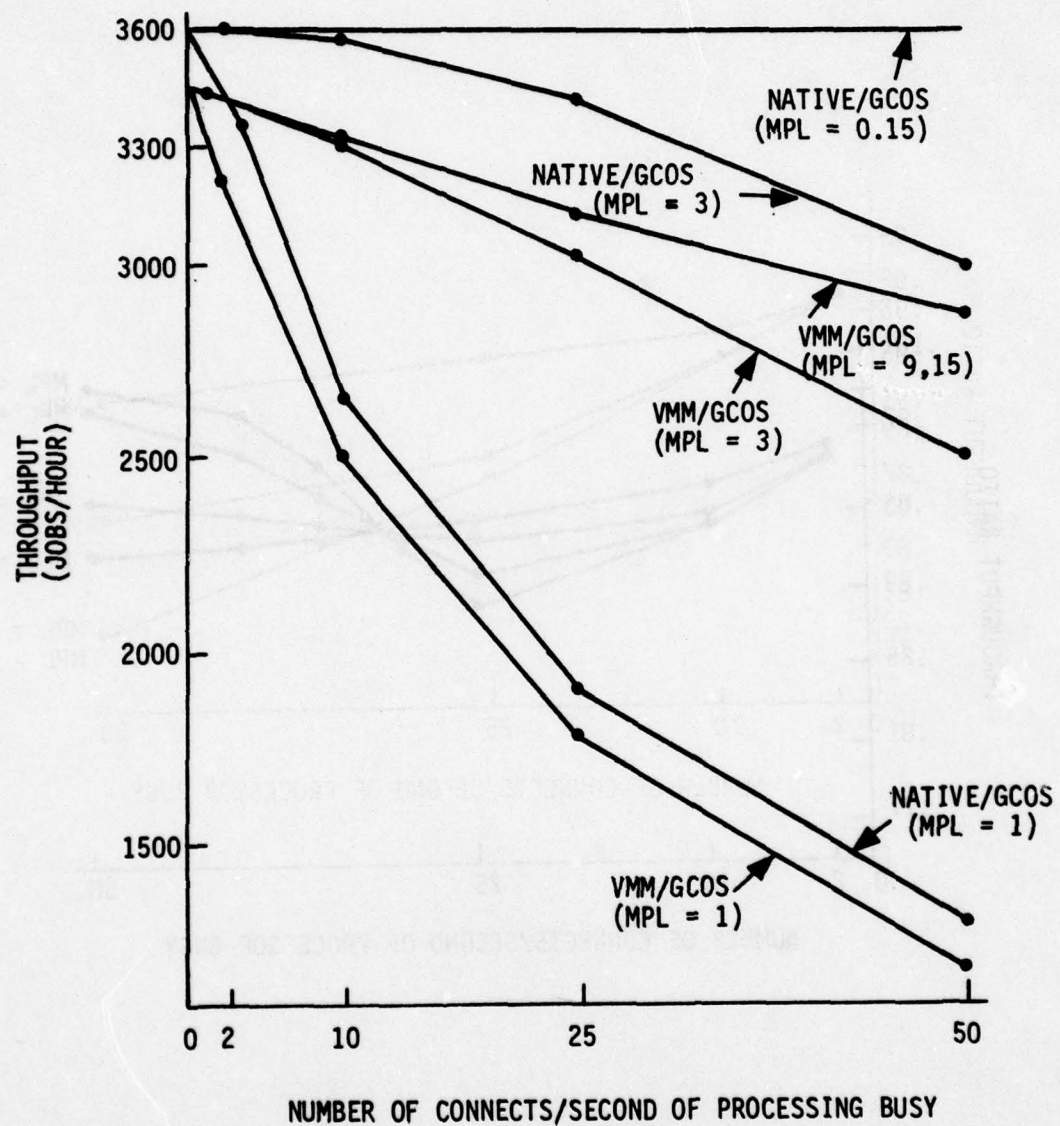


Figure 11. Effect of VMM vs. Native GCOS on Throughput as a Function of I/O Activity (I/O Contention Excluded)

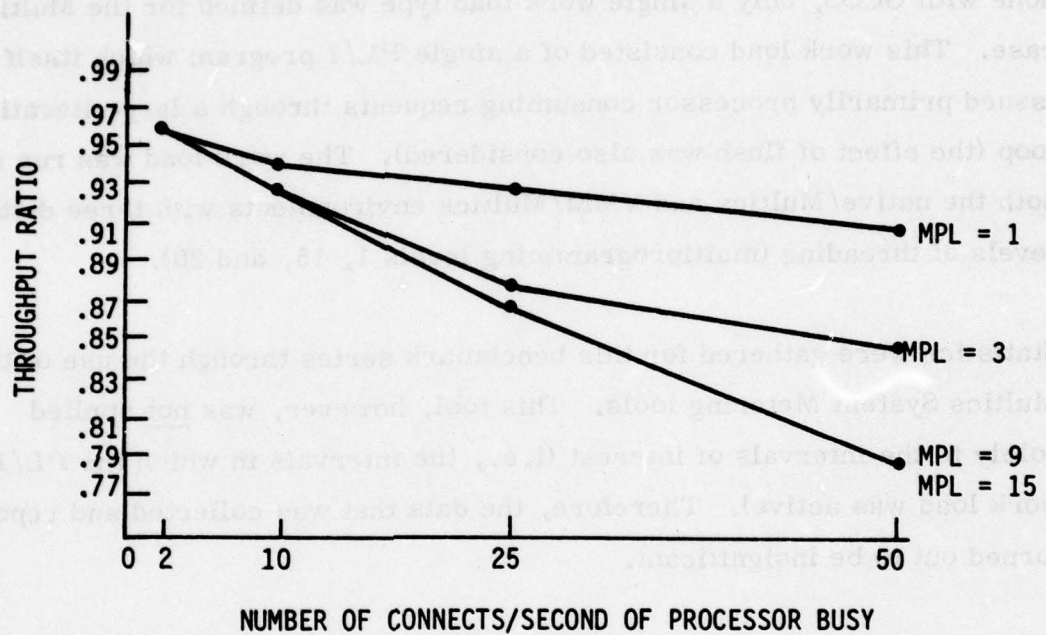


Figure 12. Ratio of VMM: Native Throughput as a Function of I/O Activity (I/O Contention Excluded)

MULTICS BENCHMARKS AND OVERHEAD ANALYSIS

The benchmark experiments run by Honeywell for Multics differ significantly from their GCOS counterparts. Rather than defining several distinct work load types and performing a series of benchmarks for each, as was done with GCOS, only a single work load type was defined for the Multics case. This work load consisted of a single PL/1 program which itself issued primarily processor consuming requests through a large iterative loop (the effect of flush was also considered). The work load was run in both the native/Multics and VMM/Multics environments with three distinct levels of threading (multiprogramming levels 1, 15, and 20).

Statistics were gathered for this benchmark series through the use of the Multics System Metering tools. This tool, however, was not applied solely to the intervals of interest (i. e., the intervals in which the PL/1 work load was active). Therefore, the data that was collected and reported turned out to be insignificant.

Fortunately, another source of information was available: a PL/1 source program referred to as LOAD-CONTROL. This program was used to monitor the work load and report the number of iterations per minute that it achieved. This provided a single data point in each environment (Table 6) which was somewhat useful in determining the VMM overhead degradation.

There are several factors which affected the usefulness of the LOAD-CONTROL data. Foremost was the fact that configurations for the native/Multics and VMM/Multics were not identical. The PL/1 work load was run under native/Multics without the use of a high-speed cache memory

and under VMM/Multics with use of the cache. Since the cache memory has a significant effect on such factors as instruction fetch rates, performance analyses using this data would clearly be affected.

TABLE 6. BENCHMARK DATA FROM THE PL/1
LOAD-CONTROL PROGRAM

	Native/Multics**	VMM/Multics*
Number of Iterations/min Single Thread	3.58	3.15

*With high-speed memory cache.

**Without high-speed memory cache.

Even so, a first-order analysis of the data was attempted. Based on the above data, it was deduced that under the native/Multics environment 16.76 seconds were required per iteration while under VMM/Multics this number increased to 19.05 seconds per iteration; this implied an observed VMM overhead degradation of 13.7 percent.*

*This represents an increase factor of 4 over the degradation for the processor busy contribution in the GCOS use. These numbers are not intended to be compared, however, Paging activity, for example, which is not present in the GCOS environment, causes a significant performance impact in the Multics environment. The effect of Multics paging activity has in effect been aggregated into the Multics processor busy contribution. Explicitly determining the degradation contributed by paging would require additional benchmark experiments and substantial further investigation.

Since in the benchmark experiment for the VMM/Multics case the cache memory was incorporated, this figure represents a lower bound on the VMM overhead degradation.

The actual degradation would clearly be more significant for larger effective iteration speedups contributed by use of the cache. Since typical speedup factors introduced by the use of a high-speed cache may range from 5 to 30 percent for processor bound work loads, this leads to the ranges of VMM overhead degradation depicted in Table 7.

TABLE 7. ESTIMATED VMM OVERHEAD DEGRADATION
AS A FUNCTION OF CACHE CONTRIBUTION

Effective Cache Contribution (%)	Estimated VMM/Multics (Second/iteration without cache)	Percent Degradation VMM: Native
0	19.05	13.7
5	20.05	19.6
10	21.17	26.3
15	22.41	33.7
20	23.81	42.1
25	25.4	51.6
30	27.2	62.3

BEST/1_{tm} -MULTICS ANALYSIS

Even with the minimal amount of data provided by the benchmark experiments in the Multics case, examination of the performance impact of the VMM degradation for a variety of hypothesized configuration and work load alternatives was attempted. This was done with the aid of BGS Systems' proprietary modeling package BEST/1_{tm}.

For purposes of this analysis, the θ_2 factor in the VMM/Multics case was assumed to be the same as that in the VMM/GCOS case (0.0045 seconds/connect). The θ_1 factor was varied over the end points of the range of effective cache contributions described previously. The performance of the hypothesized systems in the native machine mode under Multics was analyzed using BEST/1_{tm}. Next the coefficients θ_1 and θ_2 were used to determine the VMM overhead degradation of the hypothesized systems under Multics in the virtual machine mode. BEST/1_{tm} was then used again to determine the performance impact of executing the hypothesized systems under VMM/Multics.

As in the BEST/1_{tm} -GCOS analysis, the configuration and work load models consisted of a canonical job stream executing on a single processor system. Two hardware configurations were modeled: one including the effects of I/O device contention, and one excluding these effects. Individual tasks in the job stream were assumed to consume approximately 1 second of processor time and to perform a varying number of I/O operations (0 to 50) consuming approximately 35 msec per connect. The analyses were performed for the job stream under several distinct levels of system load, ranging from a single load to a load level of 15.

The models chosen and analyzed via BEST/1_{tm} were directed toward determining the VMM overhead impact on two important measures of system performance: response time and system throughput. The results of the analysis are presented in Figures 13 through 16.

SUMMARY OF RESULTS

Initial benchmark experiments and measurement data, obtained for the Honeywell 6180 configuration at RADC, have provided an indication of the performance of executing job streams under the GCOS and Multics operating systems in both native and virtual machine environments. Analysis of this data through multiple linear regression and queing network techniques provided insight into the performance degradation of running work loads under GCOS and Multics in both the native and virtual machine modes. The following was observed:

- For both GCOS and Multics, both measures of system performance--response time and throughput--showed that VMM overhead had its greatest effect on work loads exhibiting an intermediate amount of I/O activity (15 to 35 connects per second of processor busy time).
- In case of GCOS, the performance impact of the VMM overhead increases with the load for processor bounded work loads and decreases with the load for I/O bounded work loads. For work loads exhibiting a small amount of I/O activity, VMM overhead has only a minor effect on performance. For work loads exhibiting a large amount of I/O activity, contention at the I/O devices becomes the limiting factor and again the VMM overhead contributes only a minor effect. If contention at the I/O devices could be

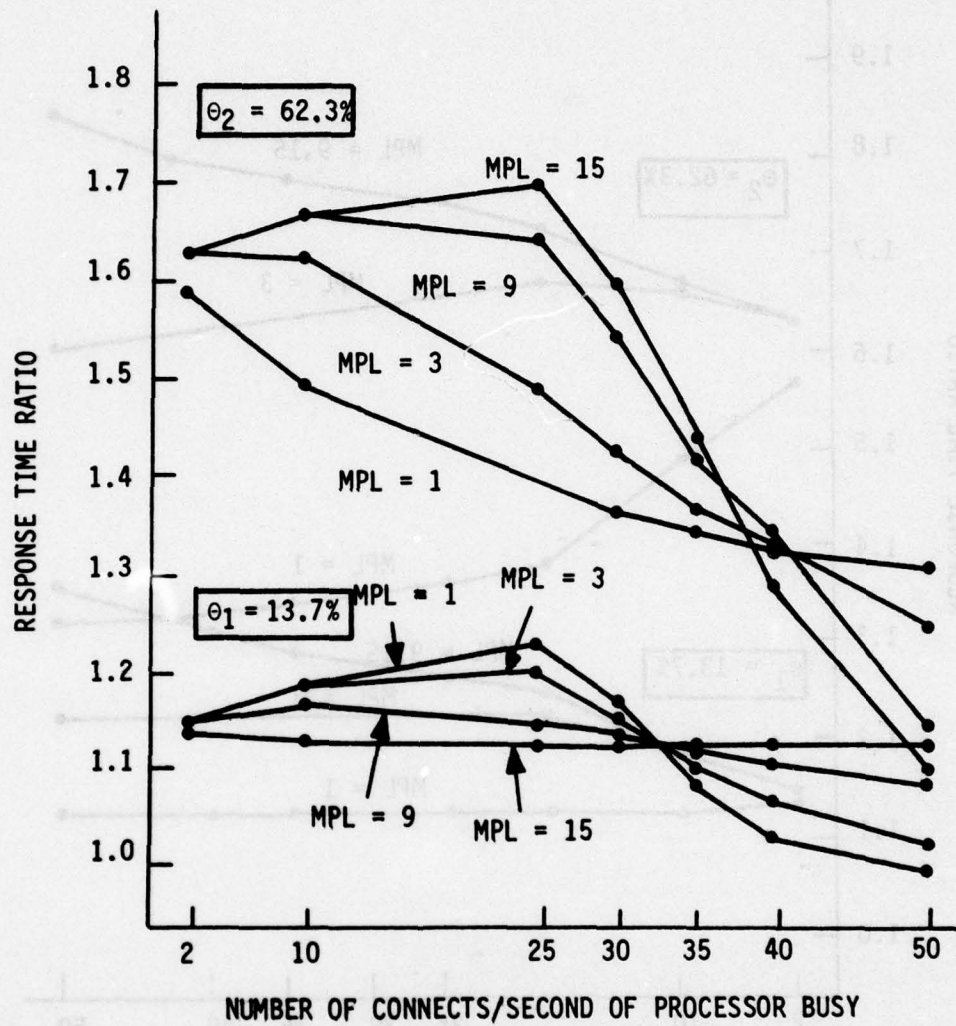


Figure 13. Ratio of VMM: Native Response Time for Multics as a Function of I/O Activity (I/O Contention Included)

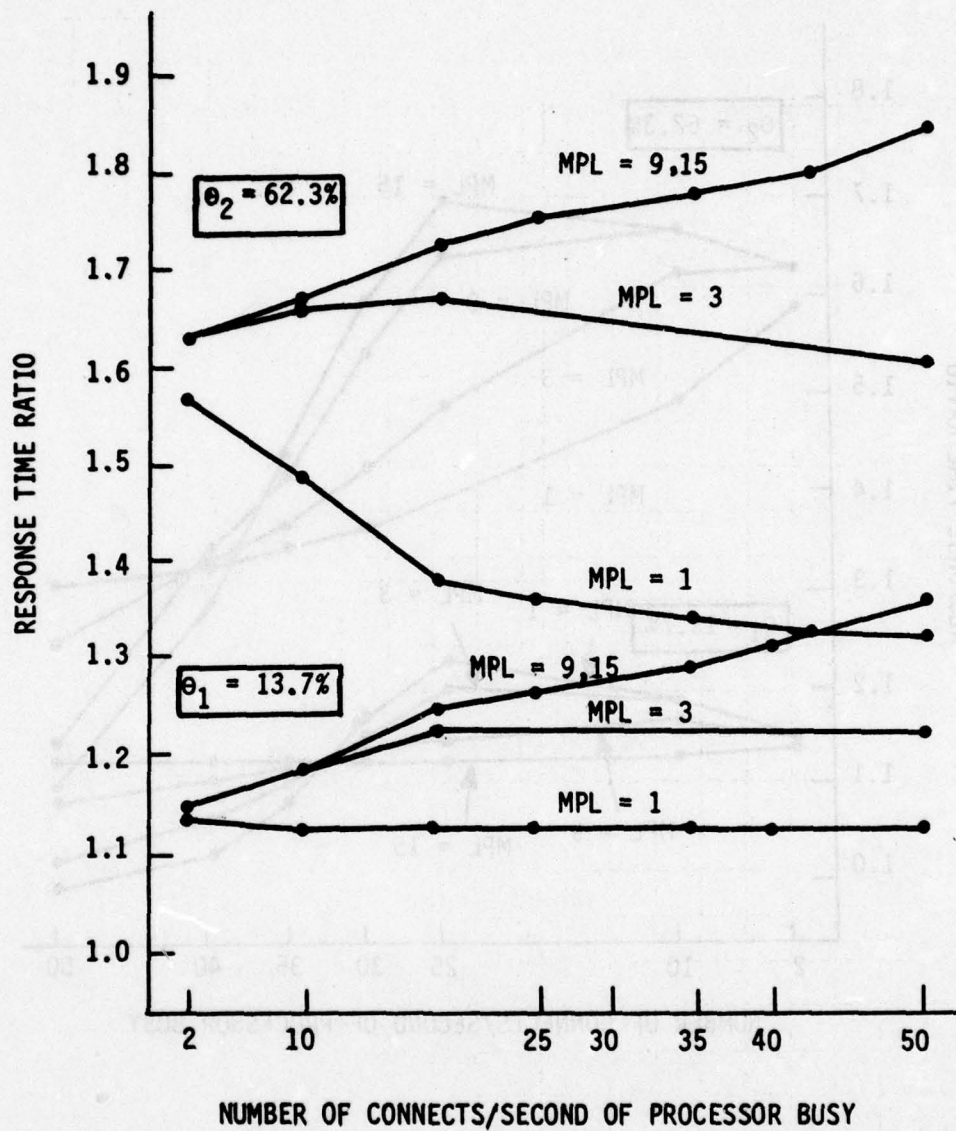


Figure 14. Ratio of VMM: Native Response Time for Multics as a Function of I/O Activity (I/O Contention Excluded)

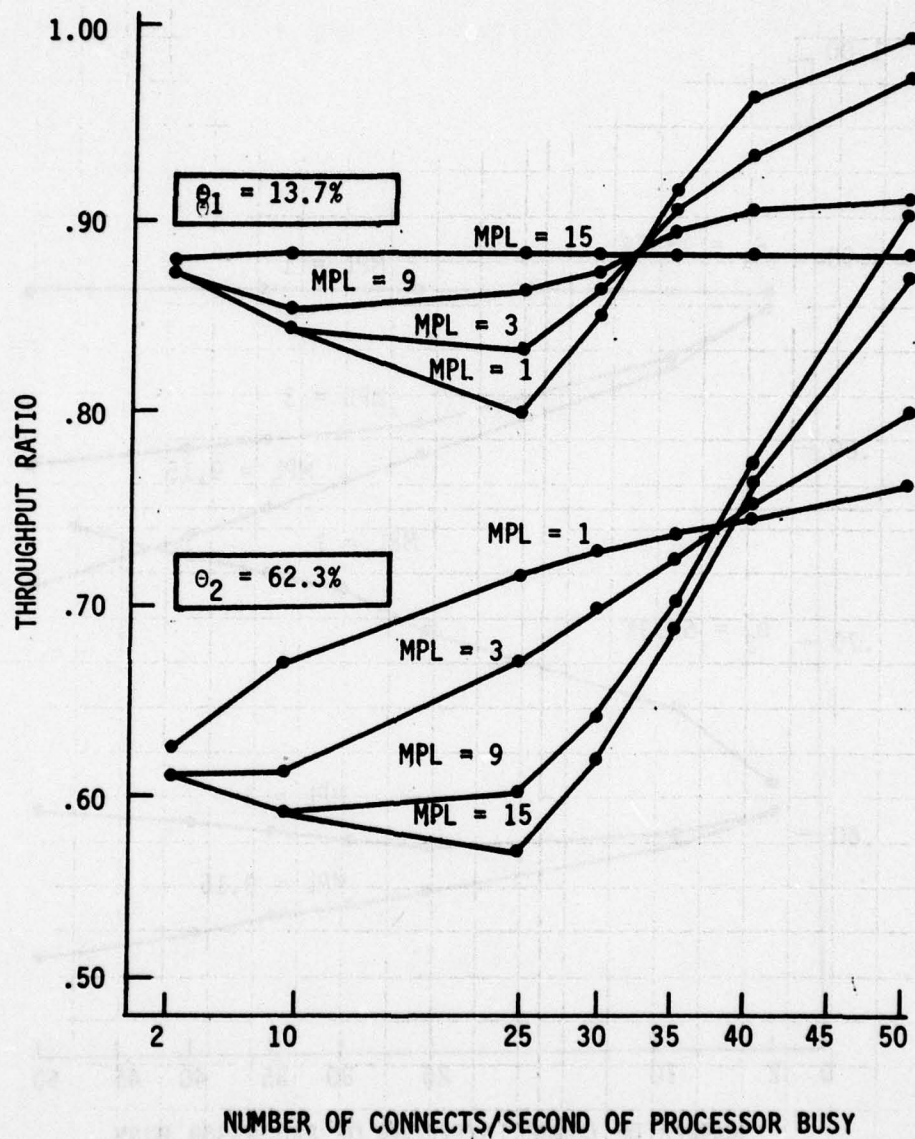


Figure 15. Ratio of VMM: Native Throughput for Multics as a Function of I/O Activity (I/O Contention Included)

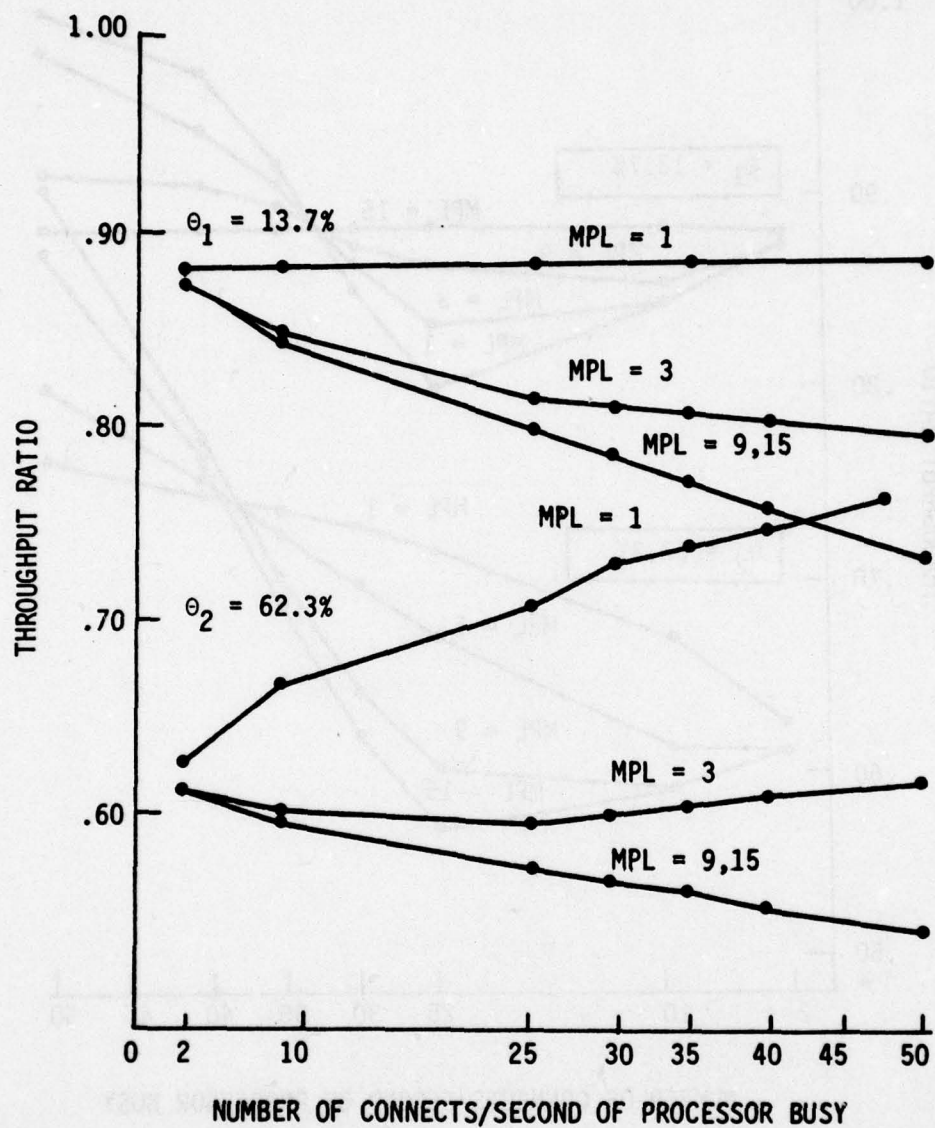


Figure 16. Ratio of VMM: Native Throughput for Multics as a Function of I/O Activity (I/O Contention Excluded)

eliminated in the latter case, however, the VMM overhead would again become a major source of performance degradation. Similar statements, although probably applicable to Multics, are not conclusive because of insufficient benchmark data in the Multics case.

- In GCOS the VMM overhead was determined to be in the range of 15 to 28 percent depending upon the degree to which the job mix was I/O dominated. The refinement of these factors in terms of processor busy time and the I/O activity is 3.5 percent and 4.5 msec of overhead per connect, respectively.
- In Multics a substantially higher value of VMM overhead was determined. The range of this value was estimated to be very wide, 13 to 60 percent, because of the insufficient number of benchmark experiments. Overhead of greater than 30 percent is likely.
- The relatively higher values for the Multics/VMM degradation vs. GCOS/VMM degradation are attributed to the higher VMM overhead required to process page faults and associated I/O within the Multics virtual machine.
- Reference to the θ_1 ($A = 2.2$ percent) and θ_2 ($B = 4.8$ msec/con) values for the "pure" work loads reflects minimum VMM overhead when GCOS is not issuing master mode instructions.

SECTION IV

EVOLUTION OF THE VMM

VMM APPLICATIONS

Early experience with the VMM has shown that the concept has a major impact on the development of operating system software. More recently, potential assistance to the application programmer has come to light as a result of VMM research. The Honeywell VMM provides a minimum set of capabilities which allow RADC to evaluate this concept in a limited environment. Some discussion of the major applications will help focus this evaluation.

Virtual Microcomputers and Minicomputers

It is convenient to describe the many levels of processing (mini, micro, macro, etc.) and the units which they use as a computational theater. Such a theater is a formally described processing system including a basic idea of computation which is independent of level. It is possible for a VMM to provide a theater which can produce multiple copies of itself. Within each theater, then, a particular instance of a processing unit could be created.

Since VMMs provide complete hardware/software interfaces, multiple simultaneous users, and fictitious I/O devices, the advantages of large machines can be extended downward for small machines. A large number

of mini or microcomputers could be encapsulated on a single large system executing a VMM. This would allow extension of the capabilities of small machines for software development, multiprogramming, and computer systems research.

Networking

Given a situation where the software for a geographically distributed system needs to be developed but only a single hardware system is available, a VMM can provide the needed test bed. By establishing multiple VMs within the VMM, each virtual machine can communicate with the outside world and other machines on the pseudo-network without actual communication lines. The information can move from one VM to another VM by passing through the VMM or by going outside the machine to a wrap-around communications device.

Program Debugging

One possible way to use the VMM approach for debugging software is to allow a virtual machine to be "smart" about its environment, that is, allow a VM to understand its interface with the VMM and to know that other VMs exist on the same host hardware. In this way a "spy" program can be executed on VM1 which views the execution of VM2, traps data about that execution, and provides a debug interface to a user of VM1. The spy concept was implemented at the IBM Grenoble Scientific Center on a modified CP-67.

Other software debugging aids using VMM concepts are extended console functions and recursion. With a virtual operator's console simulated on a user's terminal, many system commands can be extended out to the user. Examination and modification of absolute addresses and dynamic modification of system scheduling parameters are two of these. The aspect of recursion can be called nested VMMs. In this use of a VMM, a specialized debug-oriented VMM is executed under the bare machine monitor. Thus a VMM becomes a VM. Obviously, efficiency considerations may limit the depth to which recursion is feasible.

Input/Output Applications

Two major applications of VMM are I/O program analysis (virtual I/O) and new peripheral support. The first of these was discussed in the interim report and will not be elaborated upon here. The extension of the virtual I/O concept allows us to visualize three scenarios in which a VMM is used to aid the introduction of a new peripheral.

Scenario 1. A New Peripheral Device is Being Proposed or Developed for an Existing Product Line--By providing a software replica of a device which does not currently exist, virtual machine systems can be of significant value in performance evaluation studies and in software development work. Since the VMM can guarantee good program performance, it is possible using VM techniques to run highly complex test or benchmark programs to determine the quality of the software support. Since virtual machine systems usually provide good tools for evaluating performance of software running on VMs, it is easy to perform these evaluations. It also becomes easier to test new error handling routines while running on

a VM since the VMM may simulate errors on the device. If the development of new peripherals is a frequent activity of an organization, it may be possible to use a higher level language to describe the device and compile the VMM virtual I/O device support directly from this description. The VMM support for the new device might be mapped, partitioned, or simulated depending upon the device's degree of departure from existing designs.

Scenario 2. A New Peripheral is Introduced into a Computer System and There is No Software Support for It in the Commonly Used Operating System--

Virtual machines permit the introduction of a new peripheral device into a system in which the operating systems do not support the device. Thus, VMMs allow an installation to take advantage of new peripheral technology without rewriting the operating system's device support package. The VMM continues to provide the illusion of some device which the operating system already supports while in reality it uses the new device instead. Depending upon the similarities between devices, the support might be mapped, partitioned, or simulated. Since today's modern operating systems are enormously complex, it is often preferable to introduce (the actual) support for the new device in the VMM which is smaller, simpler, and easier to debug.

The technique has been used successfully in the CP-67 system. IBM 2314 disk units were first introduced into the system at the VMM level while the operating system (i. e., CMS) continued to manipulate IBM 2311s as virtual I/O devices. The "mini-disks" were supported as partitioned mapped devices.

Scenario 3. A New Peripheral is Being Introduced into a Computer System and There is No Software Support for It in the Commonly Used Operating System; However, There Exists Some Specialized Stand-alone Software Which Does Support the Device--Suppose it is necessary to support a new device which neither the predominate operating system nor the VMM is able to support. Suppose further that some other special purpose operating system is able to support that device. If standard techniques exist for communicating information between two virtual machines running under the same VMM, it is possible for the predominant operating system to use the device by sending access requests to the special purpose operating system via the VMM's communication mechanism. Thus no changes have to be made to any of the systems. Furthermore, the process of debugging the device handling routines in the special operating system can only affect that system and cannot cause the predominant operating system to crash. These techniques were used very effectively by MIT Lincoln Laboratory in debugging and introducing support for the ARPANET in the CP-67 system.

A similar scenario arises when it is necessary to run test and diagnostic software for a new device before that software has been integrated into the commonly used operating systems. In this case the standard operating system runs on one virtual machine while the Test and Diagnostic Monitor runs on another. There is no need to communicate between virtual machines in this example since the (stand-alone) Test and Diagnostic Monitor is controlled through the virtual operator's console of the VM it is running on.

SOFTWARE EXTENSIONS: THE SERVICE MACHINE

The VMM under evaluation in this effort represents an implementation of functionality which was known to be a partial fulfillment of the goals of a long-term project. Therefore, there are several areas of functional extension which are known to be desirable from several points of view. Among these are extensions for sharing peripheral equipment, including front end processors, enhanced system console functionality to permit dynamic changes in virtual machines being run and their real resource assignments, an ability to dynamically swap virtual machines so that several can be multiplexed, improvements to the treatment of timer and clock information so that each virtual machine can have a correct time-of-day, and implementation of the service machine concept as a vehicle to greatly enhance the VMM functionality for those services which can tolerate the internal delays associated with dispatching a virtual machine (i. e., the service machine).

The main recommendations of software issues which result from this analysis are that:

- Performance degradation due to input/output operations is substantial as discussed above and needs careful analysis in the design of this or any other VMM.
- The areas of missing functionality identified above are important in order for a VMM to be of practical value for most areas of application.

In this section we present an implementation approach for the virtual machine monitor. The approach derives from desires to keep the permanently resident VMM code as small as possible and to make use of already existing software. In particular, much of the functionality required within the VMM can be found in either the Multics or GCOS operating system. Two important examples of this necessary functionality are an interactive user interface and a named information storage system.

Figure 17 is one example of a VMM implemented using Multics to provide support for the resident part of the VMM. This figure shows a stylized view of a computer system memory. There is a permanently resident area which is part of the VMM. The Multics operating system is running in one virtual machine, and two different GCOS operating systems are operating independently in two additional virtual machines.

Within the Multics virtual machine there are many processes running which, in this example, fall into two categories: service machine processes (SMP) and user processes. The SMPs comprise the rest of the VMM beyond the resident part of the VMM (RVMM). In general there is one SMP per active virtual machine whether or not that VM is physically resident in primary memory. Further, there are additional SMPs working on behalf of the RVMM to perform overall functions for the VMM (e.g., VMM operator console, I/O spooling activities, etc.).

Within the Multics virtual machine are also shown user processes. These are shown only to suggest that the Multics system could be used to support normal Multics service for a user community in addition to the SMPs.

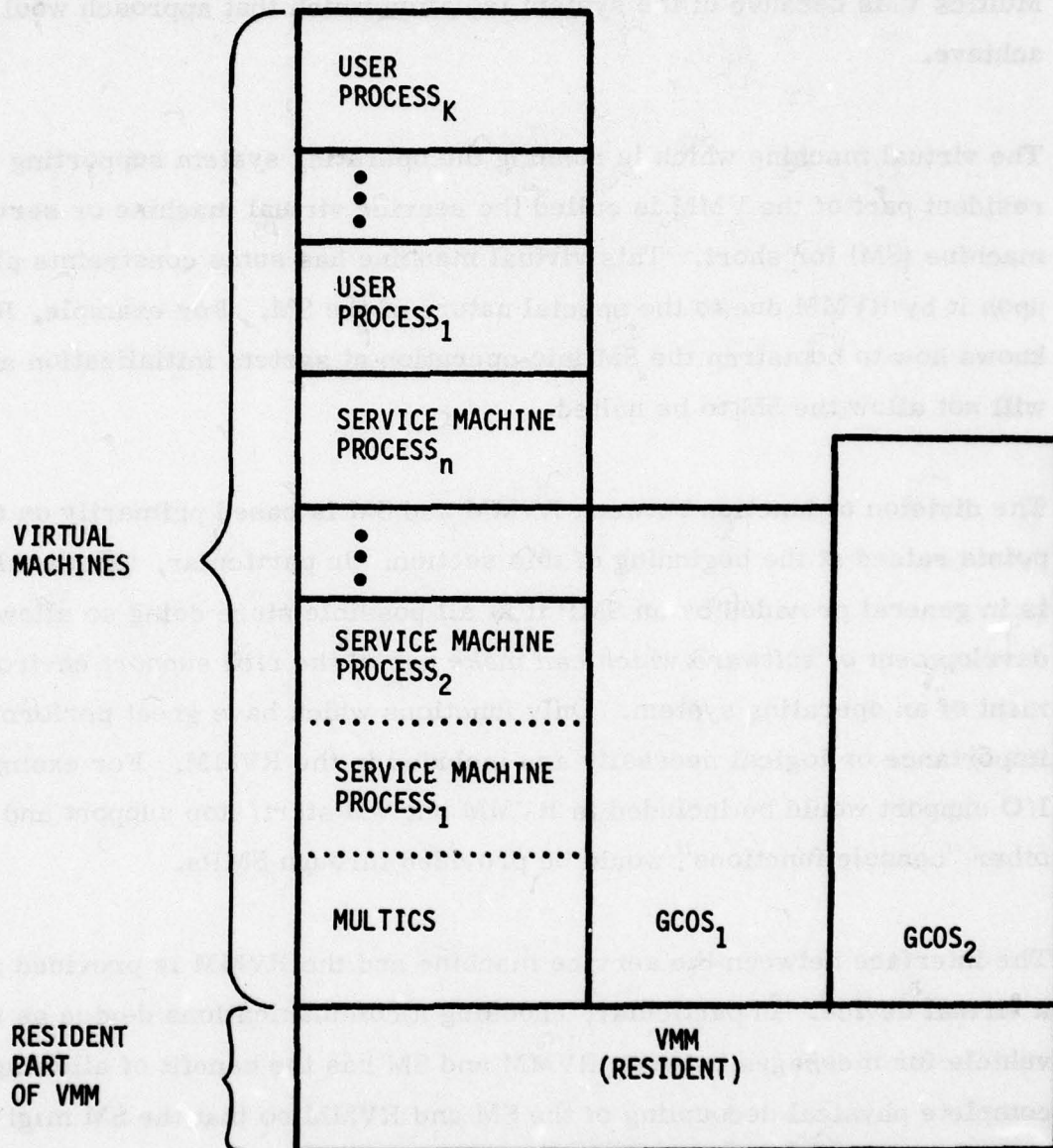


Figure 17. Service Machine Memory Layout

However, some system usage environments might choose to run two Multics VMs because of the system isolation which that approach would achieve.

The virtual machine which is running the operating system supporting the resident part of the VMM is called the service virtual machine or service machine (SM) for short. This virtual machine has some constraints placed upon it by RVMM due to the special nature of the SM. For example, RVMM knows how to bootstrap the SM into operation at system initialization and will not allow the SM to be halted.

The division of function between RVMM and SM is based primarily on the points raised at the beginning of this section. In particular, functionality is in general provided by an SMP if at all possible since doing so allows development of software which can make use of the rich support environment of an operating system. Only functions which have great performance importance or logical necessity are included in the RVMM. For example, I/O support would be included in RVMM but VM start/stop support and other "console functions" would be provided through SMPs.

The interface between the service machine and the RVMM is provided as a virtual device. In particular, choosing a communications device as the vehicle for messages between RVMM and SM has the benefit of allowing complete physical decoupling of the SM and RVMM so that the SM might be supported on a remote computer.

RVMM sends messages to SM by simulating an interrupt to the virtual machine running SM for the appropriate communications line and device. Likewise, SM sends messages to RVMM using the normal operating system I/O code for the communications device and line which is appropriate. The RVMM recognizes this I/O request and directly consumes the message.

HARDWARE EXTENSIONS

The main hardware issues in the design of any VMM relate to the question of how much of the system's overall resources are used in supporting virtual machines. Each real resource of a computer system (processor, memory, devices, channels, switches) needs to be replicated for each virtual machine. To do this, the hardware must either directly perform the mapping between virtual and real resources, or it must have some mechanism of intercepting references to virtual resources, capturing the complete state of the virtual machine, and passing control to some other agent (e.g., a software or firmware VMM). This agent then simulates the correct behavior of the virtual resource and returns control to the virtual machine for further execution under direct hardware control. The agent consumes real resources in performing this simulation and this effect can be substantiated.

In the VMM under study it was shown that the amount of real system resources consumed by the VMM in processing input/output operations is a significant level for normal computer system work loads. The percent of the central processor needed for this purpose ranged from 10 to 30 for normal GCOS work loads and somewhat higher for Multics.

Direct hardware support for input/output operations is the single most important potential for decreased VMM overhead. Such support would not be required for all I/O devices. A study of the device usage shows that disk and tape units have the most potential for reducing VMM overhead since I/O operations occur most frequently on these units.

Direct support of unit record equipment is not needed especially since it is desirable for this equipment to be shared by all virtual machines via direct VMM control of the device and spooling of the records.

Further investigation into the architectural approaches to communications front end processor support is necessary before the question of appropriate hardware support for virtualization can be answered. The importance of such support is, of course, dependent on the types of work loads that might be processed.

Another area in which hardware support is of great importance is that of main memory mapping. The present VMM statically allocates the full real memory required by a virtual machine. For several classes of use of virtual machines, it would be desirable to have more dynamic control of the mapping between virtual and real memory resources. In particular, a block assignment or paging mechanism could be investigated for its effect on VMM overhead for different classes of work loads.

EVOLUTION OF HONEYWELL COMPUTER PRODUCTS

The future development of Honeywell computer equipment depends upon the direction imposed by industry plus the technology available from the research and development groups both within Honeywell and in the academic community. Honeywell has a history of capitalizing on advanced development which is a result of their position as a leader in digital technology.

External Influences

The major influences on the future products come from two sources: industry direction and architectural influences. The industry needs for distributed computing power at the point of need, working on common centralized data bases, are reflected by the Distributed Systems Environment announced recently by Honeywell. This environment allows many of Honeywell's computer products to cooperate in new ways to achieve the desired goal. The Level 6 minicomputer can be used as a local batch processor, a remote job entry device, or a message coordinator in a larger network. When combined with the Level 66 GCOS machines and the Level 68 Multics system, a network of considerable flexibility can be constructed.

The architectural influences come from our understanding that computer architecture comprises the rules, standards, protocols, and guidelines that govern the design and development process. Also included is the user interface to such systems. Influences of technology evolution and application evolution point to a comprehensive architecture for the 80's whose characteristics must be:

- Very high performance, cost effectiveness
- Larger capacity, faster peripheral storage devices
- Utility grade availability
- Support for large distributed data bases
- High performance transaction processing
- Easy to use development and inquiry systems
- Standard communication links
- Aids for predicting and evaluating system performance

Increasingly we will see the use of minis as support to large systems, message switches, communication processors, and terminal controllers. Many of our large systems provide the needed characteristics. What is needed for the 80's is a means to interconnect them and to provide cross compatibility for user programs. This requires common and consistent interfaces.

Internal Influences

Honeywell has recognized the need of providing a continued flow of advanced concepts into the production divisions and has stimulated such research in several ways.

First, each computer division in the U.S. and Europe maintains an advanced development group in hardware and software. These groups are intimately familiar with current products and constantly seek new ways to use them and modifications which can improve performance and reliability.

Typical of the work of these advanced development groups is the VMM now at RADC. This software coupled with slightly modified hardware opened up a new avenue of applications for the Multics and GCOS systems. The VMM, though not available as a standard product, has proven influential in new products as will be shown later.

Secondly, the Systems and Research Center of the Aerospace and Defense Group has carried on an active program in distributed computing research. Their efforts have been aimed directly at the government market place for applications such as flight and weapon control systems, command and control systems, and certifiably secure computer programs. In fact, it is SRC that continued the VMM work represented in this document.

Third, it is recognized that computer architectures of the 1985 to 1990 time frame will require significant research and development prior to production. As a general response to that need, Honeywell formed the Corporate Computer Sciences Center in Minneapolis and directed that a program to address the architecture needs of the late 80's be started. This architecture work is responsible for interfacing with the research community, identifying promising ideas, and feeding these ideas into the planning and advanced development groups. The VMM is being considered in the course of that study.

The Role of the VMM

The virtual machine monitor was an early attempt at extending the power of the hardware/software base. Its minimal functionality is limiting in significant ways, yet the concept points to an important fact: if VMM performance is improved, the power gained should be exploited.

The advanced development processors currently on the drawing boards include a concept referred to as the hypervisor. The hypervisor is a VMM in hardware/firmware with some modifications. Honeywell has recognized that the two personalities of Level 66 and Level 68 could be put to productive use if a dual personality machine could be designed. Some general benefits of such a multicomputer or hypervisor approach are:

- Ease of moving users between systems
- Coexistence of different systems
- Expanded development and testing capabilities
- Increased capabilities with multiple copies of a limited system
- Increased availability with multiple copies
- Common interfaces
- Better test and development facilities

These benefits are directly in line with what has been learned about the existing VMM. It is conceivable that the hypervisor approach may be reflected in future Honeywell large scale processors, though changing market demands could cause this strategy to change.

SECTION V

RECOMMENDATIONS

FUTURE VMM RESEARCH

Distributed Systems of Virtual Machines

The feasibility of the virtual machine concept has been proven. Furthermore, the usefulness of supporting communication among clusters of virtual machines has been described by S.E. Madnick.¹ However, techniques and interfacing standards still need to be determined for inter-virtual machine communication and synchronization especially when the VMs are supported by physically distributed hardware systems.

Specialized Virtual Machines

In a system of logically distributed processing, it might be expected that certain systems in the network will be performing specialized functions on behalf of the other member systems. An example of this is the system which controls a data base to be shared among all member systems. The issue to be investigated is the nature of the interface between such a

¹S.E. Madnick and C. Lam, "Composite Information Systems--A New Concept in Information Systems," Report No. 35, Center for Information Systems Research, Massachusetts Institute of Technology, 1978.

specialized virtual machine and a VMM. For example, the VMM might support high level data base access primitives which completely isolate the specialized virtual machine from physical device characteristics.

Incremental System Extension and Integration

The virtual machine interface to other virtual machines in a distributed system of virtual machines is well defined and enforced by the VMM. This well defined interface might be used as the point for formal definition of a particular virtual machine function. Changes in binding between a virtual machine and the physical hardware used to support it could be made without requiring revision of the software within a virtual machine. Likewise, a cluster of virtual machines supported by one fast processor could be moved to a hardware support base of several (slower) processors without software revision.

An additional topic for investigation under this subject is the feasibility of replacing the formal interface between two particular virtual machines with a new formal interface. As a large system evolved, some virtual machines might be replaced with newer ones designed to a different formal interface. It might be useful in such situations to provide formal interface translator virtual machines which could translate between different versions of a formal interface.

CONCLUSIONS

The work performed during this effort has been valuable in that it demonstrated feasibility of a VMM in the RADC environment and identified the performance tradeoffs which must be made. The future support of this VMM by Honeywell is uncertain; however, the concept and many of the design features have been integrated into existing and planned products.

RADC has played a useful role in supporting the benefits of the virtual machine approach for certain types of problems and this work might well continue. The most fruitful area for further research appears to be in the design of a VMM which is more tolerant of I/O activity. It is doubtful that the existing prototype VMM could be improved sufficiently to allow it to function effectively in a production environment. The prototype has served its purpose of demonstrating functionality and feasibility.

BIBLIOGRAPHY

This is a partial list of references on virtual machine technology. Each of the listed references contains a bibliography which can be consulted for more detailed information.

Buzen, J.P., Chen, P.P., and Goldberg, R.P., "Virtual Machine Techniques for Improving Software Reliability," Proceedings IEEE Symposium on Computer Software Reliability, New York, 1973.

Galley, S.W., "PDP-10 Virtual Machines," Proceedings ACM SIGARCH-SIGOPS Workshop on Virtual Computer Systems, Cambridge, MA, 1973.

Galley, S.W., and Goldberg, R.P., "Software Debugging: The Virtual Machine Approach," Proceedings ACM Annual Conference, San Diego, CA, November 1974.

Goldberg, R.P., Architectural Principles for Virtual Computer Systems, PhD Thesis, Divisions of Engineering and Applied Physics, Harvard University, Cambridge, MA, 1972.

Goldberg, R.P., "Architecture of Virtual Machines," AFIPS Conference Proceedings, 1973 NCC, AFIPS Press, Montvale, NJ.

Goldberg, R.P. (ed.), Proceedings ACM SIGARCH-SIGOPS Workshop on Virtual Computer Systems, Cambridge, MA, 1973.

Goldberg, R.P., "Survey of Virtual Machine Research," Computer, June 1974.

Goldberg, R.P. and Schwenk, H.S., "Benefits of Virtual Machine Techniques for Input/Output," Infotech State of the Art Report 22-- Input/Output, Maidenhead, England, September 1974.

Goldberg, J. (ed.), Proceedings of a Symposium on "High Cost of Software", Stanford Research Institute, Menlo Park, CA, September 17-19, 1973.

IBM Virtual Machine Facility/370: Introduction, Release 3, IBM Corp.
Publication No. GC 20-1800-5, 1976.

Liuzzi, R., "L68/VMM," presentation at Forum XXII, Honeywell Large
Systems Users Association (HLSUA), Toronto, Canada, May 1976.

Madnick, S.E. and Donovan, J.J., "The Virtual Machine Approach to
Information Systems Security," IBM Systems Journal, 14, 2, May 1975.

Popek, G.J. and Kline, C., "Verifiable Secure Operating Systems
Software," AFIPS Conference Proceedings, 1974 NCC, AFIPS Press,
Montvale, NJ.

Schwenk, H.S., "Virtual Micromachines," Proceedings ACM SIGARCH-
SIGOPS Workshop on Virtual Computer Systems, Cambridge, MA, 1973.

System Metering Program Logic Manual, AN52, Revision 0, Honeywell
Information Systems Inc., Phoenix, AZ, February 1975.

Winett, J.M., "Virtual Machines for Developing Systems Software,"
Proceedings IEEE International Computer Society Conference, Boston,
MA, 1971.

APPENDIX A

JOB SCRIPTS

THIS PAGE IS BEST QUALITY PRACTICABLE
FROM COPY FURNISHED TO DDG

7386T 01 08-08-78 12.965 STAR C (=C) FILE EDITOR MAP - NEW =C FILE CONTENTS

PAGE 2

ALT # CONTENTS OF THE CHANGES ON THE NEW STAR C FILE 740704

1	S	SNUMS	CH011	
2	S	IDENT	OPERATORS, UTILITY	,55610218RADC
3	S	OPTION	FORTTRAN	
4	S	FORMAT		
5		CHARACTER	FC	
6		DIMENSION	ISTAT(2), Ibuff(320), ISTAT2(2)	
7		IOCNT	=10000	
8		IWDS	=320	
9		FC	=6H0000AA	
10	C			
11	C	FILESIZE	= 200L	
12		IFILSZ	=200	
13		LSTBLK	=(IFILSZ=12 - 1)=5	
14	C			
15		DO 10 I=1, IOCNT/2		
16		Isect=0		
17		Isect2=LSTBLK		
18		CALL GRINOS	(FC, Isect, Ibuff, IWDS, ISTAT)	
19		CALL GRINOS	(FC, Isect2, Ibuff, IWDS, ISTAT2)	
20		CALL GRROAD		
21	10	CONTINUE		
22	C			
23		CALL GRSORT		
24		STOP		
25		END		
26	S	GMAP	NDECK	
27		SYNDEF	GRINOS, GRROAD, GRSORT	
28		SYNDEF	BUSY	
29		GRINOS	NULL	
30		STXO	SVXO	
31		EAXO	2, 1=	ADDR. OF FILE CODE WORD
32		STXO	MME+2	
33		EAXO	3, 1=	ADDR. OF SECTOR NUMBER
34		STXO	SKDCW	
35		EAXO	4, 1=	ADDR. OF BUFFER
36		STXO	WRDCW	
37		LXLO	5, 1=	NO. OF WORDS
38		ANXO	=0007777, DU	SET DCW TYPE
39		SLXO	WRDCW	
40		EAXO	6, 1=	ADDR. OF STATUS WORD
41		STXO	MME+S	
42	MME	MME	OEINOS	
43		SOIA		
44		ZERO	==, SKDCW	
45		WDIC		
46		ZERO	0, WRDCW	
47		ZERO	==, 0	
48	SVXO	EAXO	==	
49		TRA	0, 1	
50	SKDCW	IOTD	==, 1	
51	WRDCW	IOTD	==, 0	

THIS PAGE IS BEST QUALITY PRACTICABLE
FROM COPY FURNISHED TO DDC

7388T 01 08-08-78 12.965 STAR C (=C) FILE EDITOR MAP - NEW =C FILE CONTENTS PAGE 3
ALT * CONTENTS OF THE CHANGES ON THE NEW STAR C FILE 740704

52	GRROAD	NULL		
53		MME	GEROAD	
54		TRA	0,1	
55	GRBORT	NULL		
56		LDQ	=3H00K,DL	
57		MME	GEBORT	
58		****		
59		*		
60	BUSY	NULL		
61		*		
62		*	BUSY WILL USE PROCESSOR TIME FOR THE NUMBER OF MILLISECONDS	
63		*	INDICATED IN THE PASSED ARGUMENT	
64		*		
65		LXL3	2,1=	GET ARGUMENT * OF MS OF BUSY TIME
66		*		
67	MSLP	NULL		
68	EAX2	100		LOOP VALUE FOR CACHE H6080
69	EAX2	77		VALUE IF NOT CACHE H6080
70		*		
71	LOOP	NULL		
72		LDQ	ONE	
73		DIV	ONE	
74		ARL	1	
75		SBX2	1,DU	
76		TPL	LOOP	
77		SBX3	1,DU	
78		TPL	MSLP	
79		TBA	0,1	RETURN
80	ONE	DEC	1	
81		END		
82	S	EXECUTE		
83	S	LIMITS	10,OK	
84	S	FILE	AA,ST1,200R	
85	S	ENDJOB		
86	S	SNUMB	CH012	
87	S	IDENT	OPERATORS,UTILITY,5581021ARADC	
88	S	OPTION	FORTTRAN	
89	S	FORTY		
90		CHARACTER	FC	
91		DIMENSION	ISTAT(2),IBUFF(320),ISTAT2(2)	
92		IOCNT	=10000	
93		IWDS	=320	
94		FC	=6H0000AA	
95	C			
96	C	FILESIZE	= 200L	
97		IFILSZ	=200	
98		LSTBLK	=(IFILSZ=12 - 1)*5	
99	C			
100		DO 10 I=1,IOCNT/2		
101		ISECT=0		
102		ISECT2=LSTBLK		

NOKEYWELL CASE PRINTING SYSTEM FILE-8

THIS PAGE IS BEST QUALITY PRACTICALLY
FROM COPY FURNISHED TO DDG

7368T 01 06-06-76 12.965 STAR C (=C) FILE EDITOR MAP - NEW =C FILE CONTENTS PAGE 4
ALT * CONTENTS OF THE CHANGES ON THE NEW STAR C FILE 740704

103		CALL GRINOS(FC, ISECT, Ibuff, Iwds, ISTAT)	
104		CALL GRINOS(FC, ISECT2, Ibuff, Iwds, ISTAT2)	
105		CALL GRROAD	
106	10	CONTINUE	
107	C		
108		CALL GRBORT	
109		STOP	
110		END	
111	S	GMAP NDECK	
112		SYNDEF GRINOS, GRROAD, GRBORT	
113		SYNDEF BUSY	
114		GRINOS NULL	
115		STX0 SVX0	
116		EAX0 2, 1*	ADDR. OF FILE CODE WORD
117		STX0 HME+2	
118		EAX0 3, 1*	ADDR. OF SECTOR NUMBER
119		STX0 SKDCW	
120		EAX0 4, 1*	ADDR. OF BUFFER
121		STX0 WRDCW	
122		LXLO 5, 1*	NO. OF WORDS
123		ANX0 =0007777, DU	SET DCW TYPE
124		SKLO WRDCW	
125		EAX0 6, 1*	ADDR. OF STATUS WORD
126		STX0 HME+5	
127	HME	HME GE INOS	
128		SDIA	
129		ZERO ** , SKDCW	
130		WDIC	
131		ZERO 0, WRDCW	
132		ZERO ** , 0	
133	SVX0	EAX0 **	
134		TRA 0, 1	
135	SKDCW	IOTD ** , 1	
136	WRDCW	IOTD ** , 0	
137	GRROAD	NULL	
138	HME	GRROAD	
139		TRA 0, 1	
140	GRBORT	NULL	
141		LDQ =3HOOK, DL	
142	HME	GRBORT	
143		****	
144		*	
145	BUSY	NULL	
146		*	
147		BUSY WILL USE PROCESSOR TIME FOR THE NUMBER OF MILLISECONDS	
148		INDICATED IN THE PASSED ARGUMENT	
149		*	
150		LXLO 2, 1*	GET ARGUMENT = # OF MS OF BUSY TIME
151		*	
152	MSLP	NULL	
153	EAX2	100	LOOP VALUE FOR CACHE H6080

MONITOR CASE PRINTING SYSTEM - P118-5

THIS PAGE IS BEST QUALITY PRACTICABLE
FROM COPY FURNISHED TO DDO

7388T 01 08-08-78 12.985 STAR C (=C) FILE EDITOR MAP - NEW =C FILE CONTENTS PAGE 5
ALT # CONTENTS OF THE CHANGES ON THE NEW STAR C FILE 740704

154	#	EAX2	77	VALUE IF NOT CACHE H6080
155	#			
156	LOOP	NULL		
157		LDQ	ONE	
158		DIV	ONE	
159		ARL	1	
160		SBX2	1, DU	
161		TPL	LOOP	
162		SBX3	1, DU	
163		TPL	HSLP	
164		TRA	0, 1	RETURN
165	ONE	DEC	1	
166		END		
167	\$	EXECUTE		
168	\$	LIMITS	10, 9K	
169	\$	FILE	AA, DP2, 2008	
170	\$	ENDJOB		
171	\$	SNUMB	CH021	
172	\$	IDENT	OPERATORS, UTILITY, 55610218RADC	
173	\$	OPTION	FORTAN	
174	\$	FORTY		
175		CHARACTER FC		
176		DIMENSION ISTAT(2), Ibuff(320), ISTAT2(2)		
177		LOCNT=10000		
178		IWDS=320		
179		FC=6H0000AA		
180	C			
181	C	FILESIZE = 2001		
182		IFILSZ=200		
183		LSTBLK=(IFILSZ=12 - 1)*5		
184	C			
185		DO 10 1=1, LOCNT/2		
186		Isect=0		
187		Isect2=LSTBLK		
188		CALL GRINGS(FC, Isect, Ibuff, IWDS, ISTAT)		
189		CALL GRINGS(FC, Isect2, Ibuff, IWDS, ISTAT2)		
190		CALL ORROAD		
191	10	CONTINUE		
192	C			
193		CALL ORSORT		
194		STOP		
195		END		
196	\$	GMAP	NDECK	
197		SYNDEF	GRINGS, ORROAD, ORSORT	
198		SYNDEF	BUSY	
199	GRINGS	NULL		
200		STX0	SVX0	
201		EAX0	2, 1#	ADDR. OF FILE CODE WORD
202		STX0	HME+2	
203		EAX0	3, 1#	ADDR. OF SECTOR NUMBER
204		STX0	SKDCW	

MONTELL CASE PRINTING SYSTEM - FILE #

THIS PAGE IS BEST QUALITY PRACTICABLE
FROM COPY FURNISHED TO DDG

73887 01 08-08-78 12.885 STAR C (=C) FILE EDITOR MAP - NEW =C FILE CONTENTS

PAGE 6

ALT * CONTENTS OF THE CHANGES ON THE NEW STAR C FILE 740704

206	EAX0	4,1=	ADDR. OF BUFFER
207	STX0	NRDCW	
207	LXL0	8,1=	NO. OF WORDS
208	ANX0	=8003237,DU	SET DCW TYPE
208	SKL0	NRDCW	
210	EAX0	8,1=	ADDR. OF STATUS WORD
211	STX0	MME+8	
212	MME	SEINBS	
213	SDIA		
214	ZERO	==,SKDCW	
215	WDIC		
216	ZERO	0,NRDCW	
217	ZERO	==,0	
218	SVX0	EAX0	
219	TRA	0,1	
220	SKDCW	==,1	
221	NRDCW	IOTD	
222	ORROAD	NULL	
223	MME	ORROAD	
224	TRA	0,1	
225	ORSORT	NULL	
226	LDQ	=3HOCK,DL	
227	MME	ORSORT	
228	----		
229	*		
230	BUSY	NULL	
231	*		
232	*	BUSY WILL USE PROCESSOR TIME FOR THE NUMBER OF MILLISECONDS	
233	*	INDICATED IN THE PASSED ARGUMENT	
234	*		
235	*	LXL3	2,1= GET ARGUMENT = * OF MS OF BUSY TIME
236	*		
237	HSLP	NULL	
238	EAX2	100	LOOP VALUE FOR CACHE H6080
239	EAX2	77	VALUE IF NOT CACHE H6080
240	*		
241	LOOP	NULL	
242	LDQ	ONE	
243	DIV	ONE	
244	ARL	1	
245	SBX2	1,DU	
246	TPL	LOOP	
247	SBX3	1,DU	
248	TRL	HSLP	
249	TRA	0,1	RETURN
250	ONE	DEC	1
251	END		
252	EXECUTE		
253	LIMITS	10,SK	
254	FILE	AA,ST1,200R	
255	ENDJOB		

THIS PAGE IS BEST QUALITY PRACTICABLE
FROM COPY FURNISHED TO DDC

7386T 01 08-08-78 12.965 STAR C (=C) FILE EDITOR MAP - NEW =C FILE CONTENTS PAGE 7
ALT * CONTENTS OF THE CHANGES ON THE NEW STAR C FILE 740704

256	S	SNUMB	CH022	
257	S	IDENT	OPERATORS, UTILITY	,55810218ADC
258	S	OPTION	FORTRAN	
259	S	FORTY		
260		CHARACTER	FC	
261		DIMENSION	ISTAT(2),IBUFF(320),IS*AT2(2)	
262		IOCNT	=10000	
263		IWDS	=320	
264		FC	=6H0000AA	
265	C			
266	C	FILESIZE	= 200L	
267		IFILSZ	=200	
268		LSTBLK	=(IFILSZ=12 - 1)*5	
269	C			
270		DO	10 1=1,IOCNT/2	
271		ISECT	=0	
272		ISECT2	=LSTBLK	
273		CALL	GRINOS(FC,ISECT,IBUFF,IWDS,ISTAT)	
274		CALL	GRINOS(FC,ISECT2,IBUFF,IWDS,ISTAT2)	
275		CALL	GRRDAD	
276	10		CONTINUE	
277	C			
278		CALL	GRBORT	
279		STOP		
280		END		
281	S	GMAP	NDECK	
282		SYNDEF	GRINOS, GRRDAD, GRBORT	
283		SYNDEF	BUSY	
284		GRINOS	NULL	
285		STX0	SVX0	
286		EAX0	2,1*	ADDR. OF FILE CODE WORD
287		STX0	MME+2	
288		EAX0	3,1*	ADDR. OF SECTOR NUMBER
289		STX0	SKDCW	
290		EAX0	4,1*	ADDR. OF BUFFER
291		STX0	WRDCW	
292		LXLO	5,1*	NO. OF WORDS
293		ANX0	=0007777,DU	SET DCW TYPE
294		SKLO	WRDCW	
295		EAX0	5,1*	ADDR. OF STATUS WORD
296		STX0	MME+3	
297	MME	MME	OEINOS	
298		SDIA		
299		ZERO	==,SKDCW	
300		WDIC		
301		ZERO	0,WRDCW	
302		ZERO	==,0	
303	SVX0	EAX0	==	
304		TRA	0,1	
305	SKDCW	IOTD	==,1	
306	WRDCW	IOTD	==,0	

HONEYWELL CASE PRINTING SYSTEM - FILE-8

THIS PAGE IS BEST QUALITY PRACTICABLE
FROM COPY FURNISHED TO DDG

7388T 01 08-08-78 12.965 STAR C (=C) FILE EDITOR MAP - NEW =C FILE CONTENTS

PAGE 8

ALT * CONTENTS OF THE CHANGES ON THE NEW STAR C FILE 740704

307	GRROAD	NULL		
308	HME	GEROAD		
309	TRA	0,1		
310	GRBORT	NULL		
311	LDQ	=3HOCK,DL		
312	HME	GERBORT		
313	****			
314				
315	BUSY	NULL		
316	*			
317	*	BUSY WILL USE PROCESSOR TIME FOR THE NUMBER OF MILLISECONDS		
318	*	INDICATED IN THE PASSED ARGUMENT		
319	*			
320		LXL3	2,1*	GET ARGUMENT = * OF MS OF BUSY TIME
321	*			
322	MSLP	NULL		
323		EAX2	100	LOOP VALUE FOR CACHE H8080
324	*	EAX2	77	VALUE IF NOT CACHE H8080
325	*			
326	LOOP	NULL		
327		LDQ	ONE	
328		DIV	ONE	
329		ARL	1	
330		SBX2	1,DU	
331		TPL	LOOP	
332		SBX3	1,DU	
333		TPL	MSLP	
334		TRA	0,1	RETURN
335	ONE	DEC	1	
336		END		
337	*	EXECUTE		
338	*	LIMITS	10,SK	
339	*	FILE	AA,DP2,200R	
340	*	ENDJOB		
341	*	SNUPB	AVDQ1	
342	*	IDENT	OPERATORS UTILITY 55810218BADC	
343	*	OPTION	FORTRAN	
344	*	FORTY		
345		CHARACTER FC		
346		DIMENSION	ISTAT(2),IBUF(850),ISTAT2(2)	
347	CCC			
348	C			
349	C	NOTE - IGCNT,IWDS CAN BE MODIFIED		
350	C	- IWDS CAN BE MODIFIED IF ISTRK IS MODIFIED		
351	C	- IFILSZ CAN BE MODIFIED IF THE FILE SIZE IN JCL IS MODIFIED		
352	C			
353	CCC			
354		IGCNT=1250		
355		IWDS=320		
356		IWDS2=320		
357		FC=6H0000AA		

RECEIVED CASE PRINTING SYSTEM FILE

THIS PAGE IS BEST QUALITY PRACTICABLE
FROM COPY FURNISHED TO DDC

7388T 01 08-08-78 12.965 STAR C (=C) FILE EDITOR MAP - NEW =C FILE CONTENTS

PAGE 9

ALT * CONTENTS OF THE CHANGES ON THE NEW STAR C FILE 740704

358	C	IFILSZ=200	
359	C	FILE SIZE IS IN LINKS	
360	C	LSTBLK=(IFILSZ=12 -1)*5	
361	C		
362	C	WORK LOOP	
363	C	FOR EACH PASS - PROCESSOR TIME = SUM OF BUSY MS + 2MS/10	
364	C	- CHANNEL TIME = 20MS/10 (APPROXIMATELY)	
365	C		
366	C	DO 100 I=1,10CNT/2	
367	C		
368	C	BOTH ISECT AND ISECT2 MUST BE WITHIN DO LOOP	
369	C	THE SUBROUTINE GRINOS MODIFIES THESE LOCATIONS	
370	C		
371	C	ISECT=0	
372	C	ISECT2=LSTBLK	
373	C		
374	C	CALL BUSY(60)	
375	C	CALL GRINOS(FC, ISECT, IBUF, IWDS, ISTAT)	
376	C		
377	C	CALL BUSY(100)	
378	C	THIS IO SHOULD CAUSE DISK HEAD MOVEMENT	
379	C	CALL GRINOS(FC, ISECT2, IBUF, IWDS2, ISTAT2)	
380	C		
381	C	WAIT UNTIL IO DONE SO THAT CAN REUSE ISECT AND ISECT2	
382		CALL GRROAD	
383	100	CONTINUE	
384		CALL GRBORT	
385		STOP	
386	S	END	
387		GMAP NDECK	
388		SYNDEF GRINOS, GRROAD, GRBORT	
389		SYNDEF BUSY	
390	GRINOS	NULL	
391		STX0 SVX0	
392		EAX0 2,1*	ADDR. OF FILE CODE WORD
393		STX0 MME+2	
394		EAX0 3,1*	ADDR. OF SECTOR NUMBER
395		STX0 SKDCW	
396		EAX0 4,1*	ADDR. OF BUFFER
397		STX0 WRDCW	
398		LXL0 5,1*	NO. OF WORDS
399		ANX0 =0007777,DU	SET DCW TYPE
400		SXL0 WRDCW	
401		EAX0 6,1*	ADDR. OF STATUS WORD
402		STX0 MME+5	
403	MME	MME OEINOS	
404		SDIA	
405	ZERO	==,SKDCW	
406	WDIC		
407	ZERO	0,WRDCW	
408	ZERO	==,0	

HONEYWELL PAGE PRINTING SYSTEM- P1188-8

THIS PAGE IS BEST QUALITY PRACTICABLE

FROM COPY FURNISHED TO DDQ

7388T 01 08-08-78 12.965 STAR C (*C) FILE EDITOR MAP - NEW *C FILE CONTENTS

PAGE 10

ALT # CONTENTS OF THE CHANGES ON THE NEW STAR C FILE 740704

409	SVX0	EAX0	**	
410		TRA	0,1	
411	SKDCW	IOTD	** ,1	
412	WRDCW	IOTD	** ,0	
413	ORROAD	NULL		
414		HME	GERCAD	
415		TRA	0,1	
416	ORBORT	NULL		
417		LDO	=3HOOK,DL	
418		HME	GEBORT	
419	****			
420	*			
421	BUSY	NULL		
422	*			
423	*	BUSY WILL USE PROCESSOR TIME FOR THE NUMBER OF MILLISECONDS		
424	*	INDICATED IN THE PASSED ARGUMENT		
425	*			
426		LXL3	2,1*	GET ARGUMENT = # OF MS OF BUSY TIME
427	*			
428	MSLP	NULL		
429		EAX2	100	LOOP VALUE FOR CACHE H6080
430	*	EAX2	77	VALUE IF NOT CACHE H6080
431	*			
432	LOOP	NULL		
433		LDO	ONE	
434		DIV	ONE	
435		ARL	1	
436		SBX2	1,DU	
437		TPL	LOOP	
438		SBX3	1,DU	
439		TPL	MSLP	
440		TRA	0,1	RETURN
441	ONE	DEC	1	
442		END		
443	*	EXECUTE		
444	*	LIMITS	10,9K	
445	*	FILE	AA,ST1,200R	
446	*	ENDJOB		
447	*	SNUMS	AV002	
448	*	IDENT	OPERATORS,UTILITY,55810218ADDC	
449	*	OPTION	FORTRAN	
450	*	FORTY		
451		CHARACTER FC		
452		DIMENSION	1STAT(2),1BUE(550),1STAT2(2)	
453	CCC			
454	C			
455	C	NOTE - 10CNT,1WDS CAN BE MODIFIED		
456	C	- 1WDS2 CAN BE MODIFIED IF 1STBLK IS MODIFIED		
457	C	- 1FILSZ CAN BE MODIFIED IF THE FILE SIZE IN JCL IS MODIFIED		
458	C			
459	CCC			

HC NEWELL CASE PRINTING SYSTEM - P1187-9

THIS PAGE IS BEST QUALITY PRACTICABLE
FROM COPY FURNISHED TO DDG

73887 01 08-08-78 12.865 STAR C (=C) FILE EDITOR MAP - NEW =C FILE CONTENTS

PAGE 11

ALT * CONTENTS OF THE CHANGES ON THE NEW STAR C FILE 740704

460		IOCNT=3750	
461		IWDS=320	
462		IWDS2=320	
463		EC=8H0000AA	
464		IFILSZ=200	
465	C	FILE SIZE IS IN LINKS	
466		LSTBLK=(IFILSZ=12 -1)*5	
467	C		
468	C	WORK LOOP	
469	C	FOR EACH PASS - PROCESSOR TIME = SUM OF BUSY MS + 2MS/10	
470	C	- CHANNEL TIME = 20MS/10(APPROXIMATELY)	
471	C		
472		DO 100 I=1,IOCNT/2	
473	C		
474	C	BOTH ISECT AND ISECT2 MUST BE WITHIN DO LOOP	
475	C	THE SUBROUTINE GRINGS MODIFIES THESE LOCATIONS	
476	C		
477		ISECT=0	
478		ISECT2=LSTBLK	
479	C		
480		CALL BUSY(20)	
481		CALL GRINGS(FC,ISECT,IBUF,IWDS,ISTAT)	
482	C		
483		CALL BUSY(30)	
484	C	THIS IO SHOULD CAUSE DISK HEAD MOVEMENT	
485		CALL GRINGS(FC,ISECT2,IBUF,IWDS2,ISTAT2)	
486	C		
487	C	WAIT UNTIL IO DONE SO THAT CAN REUSE ISECT AND ISECT2	
488		CALL ORROAD	
489	100	CONTINUE	
490		CALL ORSORT	
491		STOP	
492		END	
493	S	OMAP NDECK	
494		SYNDEF GRINGS,ORROAD,ORSORT	
495		SYNDEF BUSY	
496	GRINGS	NULL	
497		STXO SVXO	
498		EAXO 2,1=	ADDR. OF FILE CODE WORD
499		STXO MME+2	
500		EAXO 3,1=	ADDR. OF SECTOR NUMBER
501		STXO SKDCW	
502		EAXO 4,1=	ADDR. OF BUFFER
503		STXO WRDCW	
504		LXLO 5,1=	NO. OF WORDS
505		ANXO =0007777,DU	SET DCW TYPE
506		SKLO WRDCW	
507		EAXO 6,1=	ADDR. OF STATUS WORD
508		STXO MME+5	
509	MME	MME	GEINGS
510		SDIA	

THIS PAGE IS BEST QUALITY PRACTICABLE
FROM COPY FURNISHED TO DDC

7968T 01 08-08-78 12.005 STAR C (AC) FILE EDITOR MAP - NEW AC FILE CONTENTS

PAGE 12

ALT 0 CONTENTS OF THE CHANGES ON THE NEW STAR C FILE 740704

811		ZERG	==,SKDCW	
812		WDIC		
813		ZERG	0,WRDCW	
814		ZERG	==,0	
815	SVXO	EAXO	==	
816		TRA	0,1	
817	SKDCW	IGTD	==,1	
818	WRDCW	IGTD	==,0	
819	SRGAD	NULL		
820		RNE	SRGAD	
821		TRA	0,1	
822	SEBORT	NULL		
823		LDG	=SHOCK,DL	
824		RNE	SEBORT	
825	====			
826				
827	BUSY	NULL		
828				
829			BUSY WILL USE PROCESSOR TIME FOR THE NUMBER OF MILLISECOND	
830			INDICATED IN THE PASSED ARGUMENT	
831				
832		LXL3	2,1=	GET ARGUMENT = 0 OF MS OF BUSY TIME
833				
834	MSLP	MSL1		
835		EAX2	100	LOOP VALUE FOR CACHE H8080
836		EAX2	77	VALUE IF NOT CACHE H8080
837				
838	LOOP	MSL1		
839		LDG	ONE	
840		DIV	ONE	
841		ARL	1	
842		SBX2	1,DU	
843		TPL	LOOP	
844		SBX3	1,DU	
845		TPL	MSLP	
846		TRA	0,1	RETURN
847	ONE	DEC	1	
848		END		
849	S	EXECUTE		
850	S	LIMITS	10,2K	
851	S	FILE	AA,ST1,200R	
852	S	ENDJOB		
853	S	SHUPS	AV003	
854	S	IDENT	OPERATORS UTILITY 55610216RADC	
855	S	OPTION	FORTRAN	
856	S	PORTY		
857		CHARACTER FC		
858		DIMENSION	1STAT(2),1BUE(20),1STAT2(2)	
859	CCC			
860	C			
861	C		NOTE - 10CNT,1WDS CAN BE MODIFIED	

AD-A065 087

HONEYWELL INFORMATION SYSTEMS INC MINNEAPOLIS MINN
VIRTUAL MACHINE MONITOR PERFORMANCE ANALYSIS.(U)
DEC 78 S C VESTAL , T KROCAK, H S SCHWENK

F/G 9/2

UNCLASSIFIED

F30602-77-C-0097

RADC-TR-78-251

NL

2 OF 2
ADA
065087



END
DATE
FILMED

4 -79
DDC

THIS PAGE IS BEST QUALITY PRACTICABLE
FROM COPY FURNISHED TO DDC

73867 01 08-08-78 12.985 STAR C (=C) FILE EDITOR MAP - NEW =C FILE CONTENTS

PAGE 13

ALT * CONTENTS OF THE CHANGES ON THE NEW STAR C FILE 740704

562	C	- IWD82 CAN BE MODIFIED IF LSTBLK IS MODIFIED
563	C	- IFILSZ CAN BE MODIFIED IF THE FILE SIZE IN JCL IS MODIFIED
564	C	
565	CCC	
566		ICONT=5000
567		IWD8=640
568		IWD82=320
569		FC=8M000000AA
570		IFILSZ=200
571	C	FILE SIZE IS IN LINKS
572	C	LSTBLK=(IFILSZ=12 -1)*8
573	C	
574	C	WORK LOOP
575	C	FOR EACH PASS - PROCESSOR TIME = SUM OF BUSY MS + 2MS/10
576	C	- CHANNEL TIME = 20MS/10(APPROXIMATELY)
577	C	
578	C	DO 100 I=1,ICONT/2
579	C	
580	C	BOTH ISECT AND ISECT2 MUST BE WITHIN DO LOOP
581	C	THE SUBROUTINE GRINGS MODIFIES THESE LOCATIONS
582	C	
583		ISECT=0
584		ISECT2=LSTBLK
585	C	
586		CALL BUSY(10)
587		CALL GRINGS(PC,ISECT,IBUF,IWD8,ISTAT)
588	C	
589		CALL BUSY(20)
590	C	THIS 10 SHOULD CAUSE DISK HEAD MOVEMENT
591		CALL GRINGS(PC,ISECT2,IBUF,IWD82,ISTAT2)
592	C	
593	C	WAIT UNTIL 10 DONE SO THAT CAN REUSE ISECT AND ISECT2
594		CALL ORROAD
595	100	CONTINUE
596		CALL ORSORT
597		STOP
598		END
599	S	CHAP NDECK
600		SYNDEF GRINGS,ORROAD,ORSORT
601		SYNDEF BUSY
602	GRINGS	NULL
603	STX0	SVX0
604	EAX0	2,1*
605	STX0	WMEAS
606	EAX0	3,1*
607	STX0	SKDCW
608	EAX0	4,1*
609	STX0	WRDCW
610	LXLO	8,1*
611	ANK0	=0007777,DU
612	SKLO	WRDCW

THIS PAGE IS BEST QUALITY PRACTICABLE
FROM COPY FURNISHED TO DDC

7386T 01 08-08-78 12.985 STAR C (=C) FILE EDITOR MAP - NEW =C FILE CONTENTS PAGE 14

ALT # CONTENTS OF THE CHANGES ON THE NEW STAR C FILE 740704

613		EAX0	6,1*	ADDR. OF STATUS WORD
614		STX0	MME+5	
615	MME	MME	GEINOS	
616		S01A		
617		ZERO	** ,SKDCW	
618		WDIC		
619		ZERO	0,WRDCW	
620		ZERO	** ,0	
621	SVX0	EAX0	**	
622		TRA	0,1	
623	SKDCW	IOTD	** ,1	
624	WRDCW	IOTD	** ,0	
625	ORROAD	NULL		
626		MME	GEROAD	
627		TRA	0,1	
628	ORROBT	NULL		
629		LDQ	=3HOOK,DL	
630		MME	ORROBT	
631	****			
632				
633	BUSY	NULL		
634				
635			BUSY WILL USE PROCESSOR TIME FOR THE NUMBER OF MILLISECONDS	
636			INDICATED IN THE PASSED ARGUMENT	
637				
638		LXL3	2,1*	GET ARGUMENT = * OF MS OF BUSY TIME
639				
640	MSLP	NULL		
641		EAX2	100	LOOP VALUE FOR CACHE H0080
642		EAX2	77	VALUE IF NOT CACHE H0080
643				
644	LOOP	NULL		
645		LDQ	ONE	
646		DIV	ONE	
647		ARL	1	
648		SBX2	1,DU	
649		TPL	LOOP	
650		SBX3	1,DU	
651		TPL	MSLP	
652		TRA	0,1	RETURN
653	ONE	DEC	1	
654		END		
655		EXECUTE		
656		LIMITS	10,SK	
657		FILE	AA,ST1,200R	
658		ENDJOB		
659		SHUPB	AVOOD	
660		IDENT	GENERATORS,UTILITY,SSA1021ABADG	
661		OPTION	FORTTRAN	
662		FORTY		
663		CHARACTER	FC	

THIS PAGE IS BEST QUALITY PRACTICABLE
FROM COPY FURNISHED TO DDG

7388T 01 08-08-78 12.965 STAR C (=C) FILE EDITOR MAP - NEW =C FILE CONTENTS PAGE 15
ALT * CONTENTS OF THE CHANGES ON THE NEW STAR C FILE 740704

664		DIMENSION ISTAT(2),IBUF(650),ISTAT2(2)
665	CCC	
666	C	
667	C	NOTE - IGCNT,IWDS CAN BE MODIFIED
668	C	- IWDS2 CAN BE MODIFIED IF LSTBLK IS MODIFIED
669	C	- IFILSZ CAN BE MODIFIED IF THE FILE SIZE IN JCL IS MODIFIED.
670	C	
671	CCC	
672		IGCNT=10000
673		IWDS=320
674		IWDS2=320
675		FC=SM0000AA
676		IFILSZ=200
677	C	FILE SIZE IS IN LINKS
678	C	LSTBLK=(IFILSZ=12 -1)*5
679	C	
680	C	WORK LOOP
681	C	FOR EACH PASS - PROCESSOR TIME = SUM OF BUSY MS + SMS/10
682	C	- CHANNEL TIME = 20MS/10(APPROXIMATELY)
683	C	
684		DO 100 I=1,IGCNT/2
685	C	
686	C	BOTH ISECT AND ISECT2 MUST BE WITHIN DO LOOP
687	C	THE SUBROUTINE GRINGS MODIFIES THESE LOCATIONS
688	C	
689		ISECT=0
690		ISECT2=LSTBLK
691	C	
692		CALL BUSY(8)
693		CALL GRINGS(FC,ISECT,IBUF,IWDS,ISTAT)
694	C	
695		CALL BUSY(10)
696	C	THIS 10 SHOULD CAUSE DISK HEAD MOVEMENT
697		CALL GRINGS(FC,ISECT2,IBUF,IWDS2,ISTAT2)
698	C	
699	C	WAIT UNTIL 10 DONE SO THAT CAN REUSE ISECT AND ISECT2
700		CALL ORROAD
701	100	CONTINUE
702		CALL ORSORT
703		STOP
704		END
705	S	MAP NDECK
706		SYNDEF GRINGS,ORROAD,ORSORT
707		SYNDEF BUSY
708	GRINGS	MULL
709		STXD SVXD
710		EAXD 2,1=
711		STXD MDEAR
712		EAXD 3,1=
713		STXD SKDCW
714		EAXD 4,1=

ADDR. OF FILE CODE WORD
ADDR. OF SECTOR NUMBER
ADDR. OF BUFFER

THIS PAGE IS BEST QUALITY PRACTICABLE
FROM COPY FURNISHED TO DDG

7308T 01 06-08-78 12.008 STAR C (=C) FILE EDITOR MAP - NEW =C FILE CONTENTS PAGE 16
ALT * CONTENTS OF THE CHANGES ON THE NEW STAR C FILE 740704

715	STXO	WRDCW	
716	LXLO	8,1*	NO. OF WORDS
717	ANXO	=0007777,DU	SET DCW TYPE
718	EXLO	WRDCW	
719	EAXO	8,1*	ADDR. OF STATUS WORD
720	STXO	MHE+8	
721	MHE	OEINGO	
722	ADIA		
723	ZERO	==,SKDCW	
724	WDIC		
725	ZERO	0,WRDCW	
726	ZERO	8,0	
727	SVXO	EAXO	==
728		TRA	0,1
729	SKDCW	IGTD	==,1
730	WRDCW	IGTD	8,0
731	ORROAD	NULL	
732	MHE	ORROAD	
733	TRA	0,1	
734	ORROAD	NULL	
735	LDQ	=SHOCK,DL	
736	MHE	ORROAD	
737	====		
738			
739	BUSY	NULL	
740	*		
741	*	BUSY WILL USE PROCESSOR TIME FOR THE NUMBER OF MILLISECDS	
742	*	INDICATED IN THE PARGED ARGUMENT	
743	*		
744	*	LXLS	8,1*
745	*		GET ARGUMENT = * OF MS OF BUSY TIME
746	*	NULL	
747	EAXE	100	LOOP VALUE FOR CACHE H8080
748	EAXE	77	VALUE IF NOT CACHE H8080
749	*		
750	LOOP	NULL	
751	LDQ	ONE	
752	DIV	ONE	
753	ARL	1	
754	SBXO	1,DU	
755	TPL	LOOP	
756	SBXO	1,DU	
757	TPL	MPLP	
758	TRA	0,1	RETURN
759	ONE	DEC	1
760	END		
761	*	EXECUTE	
762	*	LIMITS	10,8K
763	*	FILE	AA,DP2,200R
764	*	ENDJOB	
765		<<<<<< END OF FILE ON OUTPUT (GT). >>>>>>	

MULTICS JOB SCRIPTS

THIS PAGE IS BEST QUALITY PRACTICABLE
FROM COPY FURNISHED TO DDG

setup.ec 08/28/78 0935.5 edt Mon

```
&goto &ec_name
&label setup
\c                                     ***setup***
&command_line off
answer yes -bf dl absout>**
delete termination_flag
&if [not [exists segment ipm_status_seg]] &then create ipm_status_seg
&else truncate ipm_status_seg
ec enter_all_abs_req [date_time 2 minutes] [response "# of processes?"] [re
\cse "# of load iterations?"]
&quit
&-----
&label enter_all_abs_req
\c                                     ***enter_all_abs_req***
scl 500
ear absin>load_overseer_([index_set &2]) -hf -tm "&1" -of absout>load_overs
\c([index_set &2]) -ag ([index_set &2]) &3
scl
termination_overseer &2 &3
&quit
```

r 935 0.235 0.462 21

load_overseer_prototype.absin 08/28/78 0937.1 edt Mon

```
cwd>udd>5550c1804>Vestal>lt
load_control &1 &2
&quit
```

r 937 0.104 0.420 20

load.pll 08/28/78 0930.7 edt Mon

load: proc:

dcl a(1024) fixed bin(35);
dcl (j, k, sum) fixed bin(35);
dcl flush entry;

call flush;
do j = 1 to 100;
do k = 1 to 1024;
a(k) = 12345;
sum = sum + a(k);
end;
end;

end;

load_control.pll 08/28/78 0930.7 edt Mon

load_control: proc:

dcl ap ptr;
dcl ascii_load_id char(2) based(ap);
dcl ascii_iteration_cutoff char(4) based(ap);
dcl clock_entry returns (fixed bin(7));
dcl code fixed bin(35);
dcl cu_sarg_ptr entry (fixed bin, ptr, fixed bin, fixed bin(35));
dcl finish_time fixed bin(7);
dcl fixed builtin;
dcl float builtin;
dcl get_wdir_entry returns (char(168));
dcl hcs_initiate entry (char(*), char(*), char(*), fixed bin(1), fixed bin
\c ptr, fixed bin(35));
dcl hcs_make_seq entry (char(*), char(*), char(*), fixed bin(5), ptr, fixe
\cn(35));
dcl ioa_entry options(variable);
dcl ioa_srs entry options(variable);
dcl ipm float bin;
dcl ipm_status_seq(32) char(8) based(status_ptr);
dcl iteration_count float bin;
dcl iteration_cutoff float bin;
dcl len fixed bin(17);
dcl load entry;
dcl load_id fixed bin(35);
dcl null builtin;
dcl start_time fixed bin(7);
dcl status_ptr ptr;
dcl substr builtin;
dcl termination_flag_ptr ptr;
dcl total_minutes float bin;
dcl total_time fixed bin(7);

call cu_sarg_ptr (1, ap, len, code);
if code ^= 0 then do;
call ioa_ ("Problem with argument 1. Abort.");
call hcs_make_seq ((get_wdir()), "termination_flag", "", 10, te
\cation_flag_ptr, code);
return;
end;


```

load_id = fixed(substr(ascii_load_id, 1, len), 35);
call cu_sarg_ptr (2, ap, len, code);
if code ^= 0 then do;
    call ioa_ ("Problem with argument 2. Abort.");
    call hcs_$make_seq ((get_wdir()), "termination_flag", "", 10, te
\cation_flag_ptr, code);
    return;
end;

iteration_cutoff = float(substr(ascii_iteration_cutoff, 1, len), 27);

total_time = 0;
total_minutes = 0;
status_ptr = null;

call hcs_$initiate ((get_wdir()), "ipm_status_seq", "", 0, 0, status_
\c code);
if status_ptr = null then do;
    call ioa_ ("No ipm_status_seq. Abort.");
    return;
end;

call load;

do iteration_count = 1.0 to iteration_cutoff by 1.0;
    start_time = clock();
    call load;
    finish_time = clock();
    total_time = total_time + (finish_time - start_time);
end;

ipm_status_seq(load_id) = "finished";
termination_flag_ptr = null;

do while (termination_flag_ptr = null);
    call load;
    call hcs_$initiate ((get_wdir()), "termination_flag", "", 0, 0,
\cation_flag_ptr, code);
end;

total_minutes = float(total_time, 27)/600000000.;
ipm = iteration_cutoff/total_minutes;
call ioa_$rs("^7.4f", ipm_status_seq(load_id), len, ipm);
call ioa_ ("Iterations/minute for load ^2d = ^7.4f", load_id, ipm);

end load_control;

```

APPENDIX B

MONITORING TOOLS

THIS PAGE IS BEST QUALITY PRACTICABLE
FROM COPY FURNISHED TO DDG

SNUMS = 2777T, ACTIVITY * = 01, REPORT CODE = 06, RECORD COUNT = 00402		
IDENT	Z00232-VMM,CN5-TOMK	00000010
OPTION	FORTAN	00000020
LOWLOAD		00000030
FORTY		00000040
		00000050
		00000060
NON COMMUNICATIONS REGION IS READ PERIODICALLY.		00000070
CELLS READ ARE(IN OCTAL)		00000080
206	.CRTIR = TOTAL INTERRUPTS = CBUF(1)	00000090
207	.CRTCN = TOTAL CONNECTS = CBUF(2)	00000100
216	.GROVH = OVERHEAD TIME = CBUF(9)	00000110
236	.CRIDT = IDLE TIME = CBUF(25)	00000120
241	.CRIDT+3=CURRENT SC CLOCK VALUE FROM VMM= CBUF(26)	00000130
247	.CRTWT = # OF TIMES IDLE = CBUF(34)	00000140
250	.CRTDS = TOTAL DISPATCHES = CBUF(36)	00000150
1044	.CRUSE+0=GCOS VP TIME FROM VMM=CBUF(416)	00000160
1045	.CRUSE+1=VMM PROC. TIME FROM VMM = CBUF(416)	00000170
1046	.CRUSE+2=VMM IDLE TIME FROM VMM = CBUF(417)	00000180
1047	.CRUSE+3=MULTICS VP TIME FROM VMM = CBUF(418)	00000190
		00000200
		00000210
CHARACTER DATE		00000220
INTEGER TODPUL		00000230
INTEGER ADDR5/0206/, CBUF(500), COUNT/0650/, STAT		00000240
DATA PULHRS/230400000./, PULSEC/64000./		00000250
REAL MICHRS/3600000000./, MICSEC/1000000./		00000260
REAL ITIME, IOVH, IIDT, IRTHRS, IGCSP, IVHMP, IVMMIDL, INTXP		00000270
REAL LTIME, LOVH, LIDT, LRTHRS, LGCSP, LVHMP, LVMMIDL, LMTXP		00000280
REAL CTIME, COVH, CIDT, CRTHRS, CGCSP, CVHMP, CVMMIDL, CMTXP		00000290
INTEGER CTIR, CTGN, CTWT, CTDS		00000300
INTEGER CRTINT		00000305
		00000310
INTEGER P1, P2, P3, P4, P5, P6, P7, P8		00000320
INTEGER TCN, TIR, TDS, TWT		00000330
		00000340
		00000350
		00000360
		00000370
CALL SLEEPX(30)		00000380
INITIALIZE ALL COUNTERS		00000390
		00000400
		00000410
CALL TIMEX(DATE, TODPUL)		00000420
CALL HMOV(ADDR5, CBUF, COUNT, STAT)		00000430
IF(STAT.NE.0)GOTO 200		00000440
TIME=FLOAT(TODPUL)/PULHRS		00000450
ITIR=CBUF(1)		00000460
ITCN=CBUF(2)		00000470
IOVH=FLOAT(CBUF(9))/PULSEC		00000480
IIDT=FLOAT(CBUF(25))/PULSEC		00000490
IRTINT=CBUF(26)		00000500
IF(IRTINT.NE.0)GOTO 10		00000510
CALL ROCLCK(11,12)		00000520
IRTINT=12		00000530
10	CONTINUE	00000540
CALL ADJSCG(IRTINT)		00000550
IRTHRS=FLOAT(IRTINT)/MICHRS		00000560
ITWT=CBUF(34)		00000570
ITDS=CBUF(36)		00000580

THIS PAGE IS BEST QUALITY PRACTICABLE
FROM COPY FURNISHED TO DDG

	IGCSF=FLOAT(CBUF(415))/MICSEC	00000590
	IVMMP=FLOAT(CBUF(416))/MICSEC	00000600
	IVMMIDL=FLOAT(CBUF(417))/MICSEC	00000610
	IMTXP=FLOAT(CBUF(418))/MICSEC	00000620
C	WRITE(06,1000)DATE	00000630
	WRITE(06,1010)ITIME,IIDT,IOVH,ITCN,ITIR,ITDS,ITWT	00000640
C		00000650
	WRITE(07,1100)DATE	00000660
	WRITE(07,1110)ITIME,IRTHRS,IVMMIDL,IVMMP,IGCSP,IMTXP	00000670
C		00000680
	RESET COUNTERS FOR LOOP CALCULATIONS	00000690
C		00000700
	LTIME=ITIME	00000710
	LTIR=ITIR	00000720
	LTCN=ITCN	00000730
	LOVH=IOVH	00000740
	LIDT=IIDT	00000750
	LRTINT=IRTINT	00000760
	LTWT=ITWT	00000770
	LTDS=ITDS	00000780
	LGCSP=IGCSP	00000790
	LVMMP=IVMMP	00000800
	LVMMIDL=IVMMIDL	00000810
	LMTXP=IMTXP	00000820
C		00000830
	WRITE(42,1200)DATE	00000840
	WRITE(39,1300)DATE	00000850
C		00000860
	CC	00000870
C		00000880
	CONTINUOUS LOOP TO SAMPLE HCM CELLS	00000890
C		00000900
	100 CONTINUE	00000910
	CALL SLEEPX(15)	00000920
	CALL TIMEX(15, TODPUL)	00000930
	CALL HCMOV(ADDRS,CBUF,COUNT,STAT)	00000940
	IF(STAT.NE.0)GOTO 210	00000950
C		00000960
	CTIME=FLOAT(TODPUL)/PULHRS	00000970
	CTIR=CBUF(1)	00000980
	CTCN=CBUF(2)	00000990
	COVH=FLOAT(CBUF(9))/PULSEC	00010000
	CIDT=FLOAT(CBUF(25))/PULSEC	00010010
	CRTINT=CBUF(28)	00010020
	IF(CRTINT.NE.0)GOTO 110	00010030
	CALL RDCLK(11,12)	00010040
	CRTINT=12	00010050
	110 CONTINUE	00010060
	CALL ADJSCC(CRTINT)	00010070
	CRTHRS=FLOAT(CRTINT)/MICHRS	00010080
	CTWT=CBUF(34)	00010090
	CTDS=CBUF(35)	00011000
	CGCSP=FLOAT(CBUF(415))/MICSEC	00011010
	CVMMMP=FLOAT(CBUF(416))/MICSEC	00011020
	CVMMIDL=FLOAT(CBUF(417))/MICSEC	00011030
	CMTXP=FLOAT(CBUF(418))/MICSEC	00011040
		00011050
	CALCULATE VALUES SINCE INITIALIZATION	00011060
	T1=(((CTIME-ITIME)*3600.)+.049)	00011070
	I1=T1*10.	00011080
		00011090
		00011100
		00011110
		00011120
		00011130
		00011140
		00011150
		00011160
		00011170
		00011180
		00011190
		00011200
		00011210
		00011220

THIS PAGE IS BEST QUALITY PRACTICABLE
FROM COPY FURNISHED TO DDC

T2=(F (T(CRTINT-IRTINT)/MICSEC) + .049	00001230
I2=T2*10.	00001240
T3=CIDT-I1DT + .049	00001250
I3=T3*10.	00001260
P1=(T3/T1 + .0049)*100.	00001270
P2=(T3/T2 + .0049)*100.	00001280
T4=COVH-LOVH + .049	00001290
I4=T4*10.	00001300
P3=(T4/T1 + .0049)*100.	00001310
P4=(T4/T2 + .0049)*100.	00001320
C	00001330
C	00001340
C	00001350
T5=((CTIME-LTIME)*3600.) + .049	00001360
I5=T5*10.	00001370
T6=(FLOAT(CRTINT-LRTINT)/MICSEC) + .049	00001380
I6=T6*10.	00001390
T7=CIDT-L1DT + .049	00001400
I7=T7*10.	00001410
P5=(T7/T5 + .0049)*100.	00001420
P6=(T7/T6 + .0049)*100.	00001430
T8=COVH-LOVH + .049	00001440
I8=T8*10.	00001450
P7=(T8/T5 + .0049)*100.	00001460
P8=(T8/T6 + .0049)*100.	00001470
C	00001480
C	00001490
TCN=CTCN-LTCN	00001500
TIR=CTIR-LTIR	00001510
TDS=CTDS-LTDS	00001520
TWT=CTWT-LTWT	00001530
C	00001540
WRITE(06,1010)CTIME,CIDT,COVH,CTCN,CTIR,CTDS,CTWT	00001550
WRITE(42,1210)CTIME,I1,I2,I3,P1,P2,I4,P3,P4,	00001560
* I5,I6,I7,P5,P6,I8,P7,P8,TCN,TIR,TDS,TWT	00001570
C	00001580
C	00001590
C	00001600
PROCESS DATA WRITTEN FROM VMM TO GCOS	00001610
RTIME=CTIME + (T2-T1)/3600.	00001620
B1=T2	00001630
I1=B1*10.	00001640
B2=CVMMIDL-IVMMIDL + .049	00001650
I2=B2*10.	00001660
P1=(B2/B1 + .0049)*100.	00001670
B3=CVMMPL-IVMMP + 0.049	00001680
I3=B3*10.	00001690
P2=(B3/B1 + .0049)*100.	00001700
B4=CGCSP-IGCSP + .049	00001710
I4=B4*10.	00001720
P3=(B4/B1 + .0049)*100.	00001730
B5=CHTXP-IMTXP + .049	00001740
I5=B5*10.	00001750
P4=(B5/B1 + .0049)*100.	00001760
C	00001770
C	00001780
C	00001790
CALCULATE VMM DATA SINCE LAST SAMPLE	00001800
B6=T6	00001810
I6=B6*10.	00001820
B7=CVMMIDL-LVMMIDL + 0.049	00001830
I7=B7*10.	00001840
P5=(B7/B6 + .0049)*100.	00001850
B8=CVMMPL-LVMMP + .049	00001860
I8=B8*10.	00001870
P6=(B8/B6 + .0049)*100.	00001880

THIS PAGE IS BEST QUALITY PRACTICABLE
FROM COPY FURNISHED TO DDQ

B9=CG *-LGCSP + .049	00001870
I9=B9=10.	00001880
P7=(B9/B6 + .0049)*100.	00001890
B10=CMTXP-LMTXP + .049	00001900
I10=B10=10.	00001910
P6=(B10/B6 + .0049)*100.	00001920
WRITE(07,1110)RTIME,CRTHRS,CVMMIDL,CVMMP,CGCSP,CMTXP	00001930
WRITE(39,1310)RTIME,I1,I2,P1,I3,P2,I4,P3,I5,P4,	00001940
* I6,I7,P5,I6,P6,I9,P7,I10,P8	00001950
	00001960
RESET COUNTERS FOR NEXT LOOP	00001970
	00001980
	00001990
LTIME=CTIME	00002000
LTIR=CTIR	00002010
LTCN=CTCN	00002020
LOVH=COVH	00002030
LRTINT=CRTINT	00002040
LIDT=CIDT	00002050
LTWT=CTWT	00002060
LTDS=CTDS	00002070
LVMMIDL=CVMMIDL	00002080
LVMMMP=CVMMMP	00002090
LGCSP=CGCSP	00002100
LMTXP=CMTXP	00002110
	00002120
GOTO 100	00002130
CC	00002140
CORE MOVES ERRORS	00002150
	00002160
	00002170
200 WRITE(06,1500)STAT	00002180
STOP 200	00002190
210 WRITE(06,1500)STAT	00002200
STOP 210	00002210
	00002220
	00002230
	00002240
1000 FORMAT(1H1," ACTUAL DATA FROM GCGS HCM FOR ",A6,/,	00002250
* " TIME IDLE-SEC OVERHD-SEC CONNECTS ",	00002260
* " INTERRUPTS DISPATCHES *TIMES-IDLE",///)	00002270
1010 FORMAT(1X,F6.3,F9.1,3X,F9.1,1X,I9,3X,I9,3X,I9,4X,I9)	00002280
	00002290
1100 FORMAT(1H1," ACTUAL DATA FROM VMM FOR ",A6,/,	00002300
* " RTIME SC-HOURS VMM-IDLE-SEC VMM-PROC-SEC",	00002310
* " GCS-PROC-SEC MTX-PROC-SEC",///)	00002320
1110 FORMAT(1X,F6.3,2X,F10.4,4(3X,F10.3))	00002330
	00002340
1200 FORMAT(1H1," GCGS DATA VALUES FOR DELTA TIME INTERVAL FOR ",	00002350
* A6,/,5X,"(LAPS,IDLE,OVH ARE IN 1/10 SECOND UNITS)",/,	00002360
* 1X," TIME ",10(" "), "DATA SINCE START",10(" "),3X,	00002370
* 20("-"), "DATA SINCE LAST SAMPLE",20("-"),/,	00002380
* 1X,6X," GLAPS RLAPS IDLE %R OVHD %R",	00002390
* "GLAP RLAP IDLE %R OVHD %R",	00002400
* " CONNTS INTRPT DISPAT #IDLES",///)	00002410
1210 FORMAT(1X,F6.3,2I6,2(I6,2I3),3X,2I6,2(I6,2I3),4I6)	00002420
	00002430
1300 FORMAT(1H1," VMM DATA VALUES FOR DELTA TIME INTERVALS FOR ",	00002440
* A6,/,5X,"(VIDLE,VOVHD,GCSVP,MTXVP ARE IN 1/10 SECOND UNITS)",/,	00002450
* 1X," RTIME ",13(" "), "DATA SINCE START",14(" "),3X,	00002460
* 7("-"), "DATA SINCE LAST SAMPLE",7("-"),/,	00002470
* 1X,6X," RLAPS VIDLE % VOVHD % GCSVP % MTXVP %",	00002480
* "RLAP VIDL % VOVH % GCSVP % MTXP %",///)	00002490
1310 FORMAT(1X,F6.3,I6,4(I6,I3),5X,I4,4(I6,I3))	00002500

THIS PAGE IS BEST QUALITY PRACTICABLE
FROM COPY FURNISHED TO DDQ

C	1500	FORMAT(" CORE MOVE ERROR - STATUS = ",14)	00002510
		END	00002520
		GMAP NDECK	00002530
		TTL HCMOV - HARD CARD MOVE SUBROUTINE\770210	00002540
		TTLDAT	00002550
		EDITP ON	00002560
			00002570
		SYMDEF HCMOV	00002580
		SYMDEF SLEEPX,TIMEX	00002590
		SYMDEF RDCLCK,ADJSCC	00002600
			00002610
		CALL HCMOV(FROM,TO,COUNT,STAT)	00002620
			00002630
		CALLING ARGUMENTS	00002640
		FROM = ADDRESS OF WORD WHICH CONTAINS	00002650
		ABSOLUTE STARTING ADDRESS IN 18-35	00002660
		TO = ADDRESS OF RECEIVING BUFFER	00002670
		COUNT = ADDRESS OF WORD WHICH CONTAINS	00002680
		NUMBER OF WORDS TO MOVE IN 18-35	00002690
			00002700
		STATUS RETURNS	00002710
		STAT = 0 - ALL OK	00002720
		STAT = 1 - COUNT GREATER THAN 512	00002730
		OR ZERO OR NEGATIVE	00002740
		STAT = 2 - TO+COUNT PAST USER'S CORE	00002750
		STAT = 3 - TO IS BELOW USER'S CORE	00002760
		STAT = 4 - FROM IS NOT IN HCM (<64K)	00002770
			00002780
HCMOV	SAVE	0,2,3,4,5,6,7 ENTRY-SAVE X0,X2,X3,X4,X5,X6,X7	00002790
	STA	AR SAVE A REGISTER	00002800
	EAXO	5,1* GET ADDRESS OF "STAT"	00002810
	STXO	STAT SAVE IT	00002820
	STZ	STAT,1 SET "STAT" TO ZERO	00002830
	EAXO	2,1* GET ADDRESS OF "FROM"	00002840
	LXLO	0,0 GET "FROM" - MUST BE ABSOLUTE ALREADY	00002850
	CMPXO	=0200000,DU IS FROM < 64K (IN HCM)	00002860
	TRC	ER4 NO - ERROR 4	00002870
	STXO	FROM SAVE "FROM"	00002880
	EAXO	3,1* GET ADDRESS OF "TO"	00002890
	TMI	ER3 MINUS - ERROR 3	00002900
	STXO	TO SAVE IT	00002910
	EAXO	4,1* GET ADDRESS OF "COUNT"	00002920
	LXLO	0,0 GET "COUNT"	00002930
	TMOZ	ER1 MINUS OR ZERO - ERROR 1	00002940
	CMPXO	=01001,DU CHECK "COUNT"	00002950
	TRC	ER1 TOO LARGE - ERROR 1	00002960
	STXO	COUNT SAVE "COUNT"	00002970
	LDX2	FROM GET "FROM" AGAIN	00002980
	SBAR	BAR SAVE OUR BASE ADDRESS REGISTER	00002990
	INHIB	ON DON'T GET INTERRUPTED DURING MOVE	00003000
	MME	EMM ***ENTER MASTER MODE ***	00003010
	ASX5	TO,5 FROM ABSOLUTE "TO" BY ADDING LAL	00003020
	LDA	** ,DU GET BAR	00003030
	ANA	=0777,DU GET # OF 512 BLOCKS	00003040
	ALS	9 MOVE IT OVER	00003050
	STX5	1,1C SAVE LAL	00003060
	EAX3	** ,AU FORM UPPER LIMIT	00003070
	SBX3	1,DU SUBTRACT ONE	00003080
	LDX4	TO,5 GET "TO"	00003090
	ADX4	COUNT,5 ADD IN "COUNT"	00003100
	STX3	1,1C SAVE UPPER LIMIT	00003110
	CMPX4	** ,DU PAST OUR CORE LIMITS	00003120
	TRC	ER2,5 YES - ERROR 2	00003130
	LDX3	TO,5 NO - GET ABSOLUTE "TO"	00003140

THIS PAGE IS BEST QUALITY PRACTICABLE
FROM COPY FURNISHED TO DDC

LOOP	LDX4	COUNT, 8	GET "COUNT"	00003150
	LDA	0, 2	GET DATA	00003160
	STA	0, 3	SAVE IT	00003170
	ADLX2	1, DU	BUMP "FROM"	00003180
	ADLX3	1, DU	BUMP "TO"	00003190
	SBX4	1, DU	DECREMENT "COUNT"	00003200
	TPNZ	LOOP, 8	CONTINUE UNTIL ZERO	00003210
EXIT	TSS	++1	RETURN TO SLAVE MODE	00003220
	INHIB	OFF		00003230
	LDA	AR	RESTORE A REGISTER	00003240
	RETURN	HCMOV	RETURN TO CALLER	00003250
				00003260
ER4	AGS	STAT, 1	BUMP "STAT"	00003270
ER3	AGS	STAT, 1	BUMP "STAT"	00003280
ER2	AGS	STAT, 1	BUMP "STAT"	00003290
ER1	AGS	STAT, 1	BUMP "STAT"	00003300
	TRA	EXIT		00003310
				00003320
AR	BSS	1		00003330
STAT	BSS	1		00003340
COUNT	BSS	1		00003350
TO	BSS	1		00003360
FROM	BSS	1		00003370
				00003380
				00003390
				00003400
SLEEPX	NULL			00003410
	LDO	2, 1*	GET NUMBER OF SECONDS TO SLEEP	00003420
	MPY	64000, DL	CONVERT TO PULSES	00003430
	MME	GEWAKE	GO TO SLEEP	00003440
	TRA	0, 1	RETURN	00003450
				00003460
				00003470
				00003480
TIMEX	NULL			00003490
	MME	GETIME		00003500
	STA	2, 1*	STORE DATE	00003510
	STQ	3, 1*	STORE TIME IN PULSES	00003520
	TRA	0, 1	RETURN	00003530
				00003540
				00003550
				00003560
RDCLK	NULL			00003570
	MME	.EMM		00003580
	RSCR	32		00003590
	TSS	++1		00003600
	STA	2, 1*		00003610
	STQ	3, 1*		00003620
	MME	GESNAP		00003630
	VFD	18/0, 2/2, 1/0, 15/0		00003640
	TRA	0, 1		00003650
				00003660
				00003670
				00003680
				00003690
				00003700
				00003710
				00003720
				00003730
ADJSCC	NULL			00003740
	LDA	2, 1*		00003750
	ANA	=017777777777	USE ONLY RIGHT MOST 34 BITS	00003760
	STA	TEMP		00003770
	ANA	=0100000, DU	CHECK IF MOST SIGNIFICANT BIT = 1	00003780

	TZE	POSSCC	NO, BRANCH	00003790
	LDA	1,DL	YES, SET FLAG	00003800
	STA	FLAG		00003810
	TRA	SCCEXT	DONE	00003820
POSSCC	SZN	FLAG	IF BIT = 0, CHECK IF PREVIOUSLY WAS 1	00003830
	TZE	SCCEXT	NO, BRANCH	00003840
	LDA	#0200000,DU	IF PREVIOUSLY 1, HAD A CLOCK ROLLOVER	00003850
	ASA	ADDER	ADJUST CLOCK VALUE	00003860
	STZ	FLAG	RESET FLAG	00003870
SCCEXT	LDA	TEMP	USE ADJUSTED VALUE	00003880
	ADLA	ADDER		00003890
	STA	2,1*		00003900
	TRA	0,1		00003910
TEMP	GCT	0		00003915
FLAG	GCT	0		00003920
ADDER	GCT	0		00003930
				00003940
	END			00003950
	EXECUTE			00003960
	PRIVITY			00003970
	LIMITS	,10K		00003980
	SYSOUT	07,ORG		00003990
	SYSOUT	39,ORG		00004000

THIS PAGE IS BEST QUALITY PHOTOGRAPH
FROM COPY FURNISHED TO DDC

MULTICS TOOLS--termination_overseer

termination_overseer.pll

08/28/78 0930.8 edt Mon

```

termination_overseer: proc(ascii_n_loads, ascii_n_iterations);

dcl ascii_n_loads char(*);
dcl ascii_n_iterations char(*);
dcl character_value char(7);
dcl code fixed bin(35);
dcl cv_dec_check_entry (char(*), fixed bin(35)) returns (fixed bin(35));
dcl cv_float_entry (char(*), fixed bin(35), float bin(27));
dcl fpsum float bin(27);
dcl fpval float bin(27);
dcl get_wdir_entry returns(char(168) aligned);
dcl hcs_$initiate entry (char(*), char(*), char(*), fixed bin(1), fixed bin
\c ptr, fixed bin(35));
dcl hcs_$make_seg entry (char(*), char(*), char(*), fixed bin(5), ptr, fixe
\cn(35));
dcl i fixed bin;
dcl ioa_ entry options(variable);
dcl ipm_status_seg(32) char(8) based (status_ptr);
dcl n_loads fixed bin(35);
dcl n_iterations fixed bin(35);
dcl null builtin;
dcl status_ptr ptr;
dcl substr builtin;
dcl termination_flag_ptr ptr;
dcl timer_manager_$sleep entry (fixed bin(71), bit(2));

    n_loads = cv_dec_check_ (ascii_n_loads, code);
    if code ^= 0 then do;
        call ioa_ ("Bad argument 1 input to termination_overseer. Abort.
        call hcs_$make_seg ((get_wdir()), "termination_flag", "", 10, te
\cation_flag_ptr, code);
        return;
    end;

    n_iterations = cv_dec_check_ (ascii_n_iterations, code);
    if code ^= 0 then do;
        call ioa_ ("Bad argument 2 input to termination_overseer. Abort.
        call hcs_$make_seg ((get_wdir()), "termination_flag", "", 10, te
\cation_flag_ptr, code);
        return;
    end;

    status_ptr = null;
    call hcs_$initiate ((get_wdir()), "ipm_status_seg", "", 0, 0, status_
\c code);
    if status_ptr = null then do;
        call ioa_ ("Unable to initiate ipm_status_seg. Abort.");
        call hcs_$make_seg ((get_wdir()), "termination_flag", "", 10, te
\cation_flag_ptr, code);
        return;
    end;

```



```

do i = 1 to n_loads;
  ipm_status_seg(i) = "ZZZZZZZZ";
end;

do i = 1 to n_loads;
  do while (ipm_status_seg(i) = "ZZZZZZZZ");
    call timer_manager_sleep(30, "||b");
  end;
end;

termination_flag_ptr = null;
call hcs_make_seg ((get_wdir()), "termination_flag", "", 10, termina
\c_flag_ptr, code);

call timer_manager_sleep(30, "||b");

do i = 1 to n_loads;
  do while (ipm_status_seg(i) = "finished");
    call timer_manager_sleep(30, "||b");
  end;
end;

fpsum = 0.;

do i = 1 to n_loads;
  character_value = substr(ipm_status_seg(i), 1, 7);
  call cv_float_ (character_value, code, fpval);
  fpsum = fpsum + fpval;
end;

call ioa_ ("^3/***** Thruput for ^d loads and ^d iterations is ^.4f it
\cions per minute *****^3/", n_loads, n_iterations, fpsum);

end termination_overseer;

```

r 930 1.627 1.654 44

APPENDIX C

**"VIRTUAL MACHINE MONITOR PERFORMANCE ANALYSIS:
DESIGN PLAN"**

**Interim Report
November 17, 1977**

PREFACE

This document is intended to fulfill the requirements of line item A002, Design Plan, for contract number F30602-77-C-0097, Virtual Machine Monitor (VMM) Performance Analysis. Honeywell Systems and Research Center working with BGS Systems Incorporated have produced this document based on the result of investigations performed for the above-mentioned contract, and on research in virtual machine technology done outside the contract scope.

The principal authors were S.C. Vestal, Honeywell; H. Schwenk and R. Goldberg of BGS Systems, Inc. The authors also wish to acknowledge the assistance, both written and oral, of Mr. Russell McGee and Mr. Larry Shannon of Honeywell Information Systems.

CONTENTS

Section	Page
1 INTRODUCTION	A-1
Organization of this Report	A-1
2 VMM DESCRIPTION	A-2
Functional Description of VMM	A-2
Usage Information for VMM	A-3
Resource Objectives of VMM	A-4
Design Overview of VMM	A-5
Test Specifications for VMM	A-8
Known Limitations of VMM	A-8
Functional Description of Except-proc	A-9
Design Overview of Except-proc	A-9
Functional Description of I-proc	A-23
Functional Description of f-proc	A-32
Functional Description of vp/ipr	A-34
Functional Description of vp/acv	A-34
Functional Description of absa/routine	A-36
Functional Description of Dispatch	A-38
Design Overview of dispatch	A-39
Functional Description of vp/int	A-40
Functional Description of vector/simulation	A-41
Functional Description of vm/int/test	A-43

CONTENTS (Concluded)

Section	Page
3 DESIGN APPROACH FOR VMM EXTENSION	A-45
General Areas of Functional Need to be Examined	A-46
General I/O Considerations for Virtual Machines	A-47
I/O Device Specification	A-47
I/O Program Analysis	A-48
I/O Summary	A-49
Virtual Device Support	A-50
Dedicated Support	A-50
Partitioned Support	A-50
Mapped Support	A-52
Simulated Support	A-52
VMM Involvement	A-53
Special Considerations for Front End Processor Functional Extensions	A-53
Special Considerations for System Consoles	A-55

LIST OF ILLUSTRATIONS

Figure		Page
1	Modes of Virtual Device Support	A-51

SECTION 1

INTRODUCTION

ORGANIZATION OF THIS REPORT

Since this document was produced at approximately the midpoint of the contract term, final conclusions concerning the Virtual Machine Monitor (VMM) performance cannot be drawn. Experiments are still underway at the time of this writing. This report is an attempt, therefore, to document the functional characteristics of the VMM in a way that closely parallels its observed performance.

Section 2 describes the VMM down to the module level of detail. Much of this information has been taken from the documentation produced with the VMM implementation. The reader will notice therefore that it closely follows the WELLMADE methodology documentation structure in use at the time the design was done.

Section 3 describes the design approach for evolution of the VMM. In particular virtual device support is treated in detail for shared front-end network processors and shared operator consoles.

Additional material like that in Section 3 will be supplied at the termination of the contract, following more extensive analysis of the VMM performance in live tests.

SECTION 2

VMM DESCRIPTION

FUNCTIONAL DESCRIPTION OF VMM

The VMM provides two environments within a single 6180 system. These are indistinguishable from the normal 6000 program environment and the 6180 program environment. They are referred to as virtual machines. These environments are established by interfaces fabricated of software and hardware. When these are viewed from an operating system, they are indistinguishable from the real hardware interfaces assumed by the operating system when executing on a real machine.

The VMM under consideration supports multiple virtual 6000 machines and one virtual 6180 machine. Therefore it is possible for a single 6180 system to concurrently support multiple GCOS systems and one MULTICS. Hence we have a situation in which a VMM supports multiple operating systems in much the same way that a multi-programmed operating system would support two (or more) user jobs.

The hardware interface part of the virtual environment consists of several hardware changes to the basic 6180 CPU, a change to the IOM Direct Channel and a change to the 355 DIA Control Board. The software interfaces are supplied by the VMM and are the subject of this and related documentation.

The responsibilities of the VMM are as follows:

- Partition the available real machine resources in n pieces.
One set is reserved for the use of the VMM. The remaining resources are divided to provide one set each for $n-2$ virtual 6000 and one for the virtual 6180.
- Protect each operating system, its environment and the VMM from reference by the other operating system or any of its users.
- Provide for the allocation of the real machine processing and peripheral resources to satisfy the resource needs of the virtual machine (VM).
- Provide for the orderly start-up of the VMM, the GCOS and MULTICS.

It should be noted that the partitioning of resources must apply not only to main memory but to secondary storage and peripheral devices as well. It must not be possible for any user of any operating system to reference a virtual machine other than the one it is allocated to serve. (An anomaly exists in this respect with regard to 355s in version 1.1. See Known Limitations below.)

Usage Information for VMM

To initiate VMM operation, a bootload procedure will be started by depressing the "boot" button on the master console. This will cause the VMM to be loaded from cards. The system configuration will be specified by tables internal to the start-up deck. Once the VMM has been started up, either or both of the virtual machines may be started from a console.

In this version of VMM (only) it is the user's responsibility to establish the configuration decks of the GCOS and MULTICS to be executed in such a way that they use a complementary set of the real peripheral resources of the system. Each operating system must have its own console and unit record peripherals. It must also have its own FNP if one is used and its own magnetic tape handlers and disk spindles. However, disk and tape controllers and IOM may be shared. Furthermore, the addresses of the peripheral devices used must be their real device addresses.

After an operating system has been started from a console, it will be used exclusively by that operating system. Communication with the VMM will not be possible again unless or until the VMM detects that the operating system has ceased operation. At this time, the operating system may again be started according to the initial start-up procedure.

Once a virtual machine has been started, its interfaces to its operator and its users will be identical to those provided when it is in execution on a real machine. These are described elsewhere in appropriate manuals.

Resource Objectives of VMM

Version 1.1 of VMM shall operate on 6180 systems with memories greater than or equal to 384K words in size. It requires use of processor and direct channel modifications mentioned above. It shall be capable of supporting one or two real processors. (The VMM is fundamentally capable of supporting up to four real processors, but only a single processor has been tested thus far.)

As a development target, the VMM shall reduce combined system throughput by less than 20 percent.

Design Overview of VMM

The VMM at its most gross design level consists of two parts: start-up and the VMM body.

Start-up is executed at bootload time for the purpose of loading and initializing the VMM. Once it has been executed, it is partially overlaid and not used again until the next VMM bootload process is initiated.

The VMM body consists of two major parts: exception processing and dispatching. Exception processing fields all faults and interrupts (referred to generically as exceptions) and initiates appropriate responses to them. Exceptions may cause different types of actions to take place depending upon the exception which occurred and the environment in which it occurred.

In general, exceptions provide the method used by VMM to maintain the required degree of control over the system. The exact way in which this is done is described in lower level documentation.

After exception processing is complete, the dispatcher is invoked. The dispatcher gives control to virtual machines at the appropriate times to maintain the proper operation of the virtual machines in the proper scheduling sequence. The dispatcher contains the scheduling policy of the VMM and will give control to virtual machines in accordance with this policy.

virtual machines (if at all), the dispatcher will return control in different ways. It may simply return to the last point of execution or it may do so by simulating a fault or interrupt.

The operating systems are considered to be subroutines of the dispatcher in this design overview. The dispatcher will give control to one or the other of its virtual machines at appropriate times. Once it has given control to an operating system, the dispatcher will have finished its task and will not again be invoked for execution until another fault or interrupt has been processed by exception processing. The fundamental notion of the VMM is that, aside from the detailed instructions of the system, the only interfaces that exist between the hardware and the operating system are the I/O mailboxes and the fault and interrupt vectors. In most instances, the operating system can be allowed to execute its instructions and those of its users at full speed and without VMM intervention so long as their addresses are constrained to lie exclusively within the domain of the operating system. Only in those cases where fault and interrupt vectors or I/O mailboxes are involved is it necessary for the VMM to gain control and perform functions which will prevent operating systems from interfering with each other or with the VMM. Because of this localization of control, the VMM can control its guests with relative efficiency and the VMM itself can be kept to a fairly modest size and complexity.

The CPU modifications used in support of VMM allow the VMM to relocate the address space in which each operating system executes so that each one has the illusion that its addresses start at absolute memory address zero and proceed upward to its configured maximum. Given this illusion, each operating system also has the belief that it owns and controls the real fault

and interrupt vectors of the system. This, of course, is not true. The real fault and interrupt vectors of the system are owned and controlled by the VMM.

To maintain the degree of control which is needed and to allow the operating systems to be able to operate upon their vectors in the intended manner, the VMM gains control on every fault and interrupt. This gives the VMM the opportunity to keep track of the status of all system devices which it must control on behalf of its guest operating systems. At the same time, it gives the VMM the opportunity to exercise dispatching control over its guests. If one operating system has used a processor for its fair share of time, the VMM may not return the processor to it in the event of a fault or interrupt. It may dispatch the processor to another operating system instead. Since the VMM intercepts all faults and interrupts and controls the dispatching of the processors among the virtual machines, it is necessary for the VMM to be capable of queueing faults and interrupts and to be able to simulate these in a virtual machine at the time a processor is dispatched to it.

In addition to gaining control on every fault and interrupt, the VMM must gain control at certain other key times. Chief among these is when I/O is about to be initiated (other instances are elucidated in lower level documentation). This is essential to permit the VMM to perform address translation in the channel programs corresponding to the relocation of the virtual machine within the real address space, in some instances to perform device and channel address translation and, in future versions, to permit substitution of virtual for real devices.

At the general conceptual level, the controls described above constitute the heart of the VMM. More complete and detailed descriptions are contained in lower level documentation.

Test Specifications for VMM

The VMM is targeted to operate with an overhead of less than 20 percent. This measure shall be determined by executing a certain set of GCOS jobs and a MULTICS script as separate tasks on a freestanding 6180 system and comparing their combined execution time with the same work executing under VMM. In performing the freestanding measurements, the same system resources shall be used as are used when executing under the VMM.

Note that other variations on the method of measuring performance can be used and are perhaps preferable.

Known Limitations of VMM

The dn355 shares mass storage with the operating system which it is serving. There is no way implemented in Version 1.1 which will prevent the 355 from accessing or writing on mass storage which is outside of its intended area of use. This anomaly will be repaired in future versions of VMM.

FUNCTIONAL DESCRIPTION OF EXCEPT-PROC

The exception processor, except-proc, responds to all faults and interrupts, safestores the processor conditions at the time of the exception, and calls either a fault or an interrupt handler to take appropriate action. Except-proc will be executed by any processor which is interruptable at the time an interrupt occurs. It will be executed by the processor which detects a fault if one occurs.

The exception processor is a key element of VMM because it gains control of the system at those points and times when it is essential for VMM to gain control (for example, prior to the execution of a cioc instruction in a guest operating system). Of course, it also gains control in some instances which are of no interest to the VMM, such as when a user executes a MME. In the latter instances, the VMM merely returns control to the virtual machine after the fault is processed.

Design Overview of Except-proc

All processors share a single fault vector and a single interrupt vector. The vectors contain the instruction pair, scu/tra. In order for the results of the scus to be associated with the proper processor in the event more than one processor is in a single vector in a short time interval, ad modification is used on both instructions of the vector pair. This causes the results of each safestore to be placed in a separate eight-word area and each transfer to occur to a separate location. The processor number of each processor is set in its switches and this value is stored with the scu information so that the scu results can be correlated with the stored values after the vector pair has been executed.

A scu/tra queue is used to allow each processor to safely store scu and register conditions.

A trouble fault and an execute fault cause the system to abort; all other f/i's store the scu conditions in a scu block in the scu/tra queue and then transfer to one of the tra blocks. Six lines of code are executed that store 26 words (pregs (16), regs (8), and dsbr (2)) in that same tra block, and load x4 with the address of the tra block. The last instruction in every tra block is a transfer indirect through an its pair to the fault-interrupt-intercept-module (fiim).

Fiim is "pure" code. In this code we search "back" through the queue to find the scu conditions stored by this processor. If we have had a fault in the fiim we may find more than one scu block; we continue searching until we have found the earliest, or the one that originally brought us to fiim.

From the scu and tra blocks we can find the conditions to be stored in a machine conditions block for use by f-proc or i-proc. We store these machine condition blocks in the vpdb for faults or interrupts in a vm, in the rpdb for faults in the non-fiim VMM (e.g. absa), and we do not store conditions for interrupts in the VMM.

After storing the conditions we update the queue to show that these conditions have been picked up and stored. We also verify that the queue is not in an illegal position.

When the conditions have been stored and the queue has been updated, the processor transfers to f-proc or i-proc to finish processing the fault or interrupt.

Exception processing consists of three main parts: the vectors, the scu/tra queue and fiim. These three are described below in greater detail.

Fault/Interrupt Vector Pairs--When a fault or interrupt occurs the processor enters real, absolute, master, 6100 mode and transfers to an entry in the fault/interrupt vector.

The vector pairs are set up by VMM startup. All vector pairs except for the pair corresponding to the trouble and execute faults are set up with the same two instructions:

scu scu/tal, ad

tra tra/tal, ad

Scu/tal and tra/tal are two tally words in low core that reference a location to store scu conditions and a location to transfer to, respectively. The trouble fault causes the system to abort. Its vector pair is loaded with these two instructions:

scu trb/flt/scu

tra trb/flt/tra

At trb/flt/tra is a transfer to the system abort routine. A similar pair of instructions is found in the execute fault vector pair.

The tra tra/tal, ad causes us to transfer to a set of code (six instructions) in the scu/tra queue.

Scu/tra Queue--The scu/tra queue is composed of alternating scu and tra blocks. The scu blocks are eight words long and are used for storing the scu conditions; the tra blocks are 32 words long and are used to store the registers, pointer registers, and dsbr. There are two different types of tra blocks, since some of the storage locations in the tra blocks must be on mod 16 boundaries and 40 is not a 0 mod 16 number.

Type 1 has the storage area for regs, dsbr, and six words of code

Type 2 has the storage area for regs, pregs, dsbr, and six words of code

The six lines of code for type 1 (2) are

sreg	-10(26), ic
spri	-27(19), ic
eax4	-28, ic
sdbi	- 5, ic
idbr	vmm/dbi
tra	fiim/its/ptr,*

The transfer to fiim/its/ptr,* is a transfer to the fiim. When the fiim is entered the scu and register conditions have either been stored in a valid location or they have not and the fiim will detect this.

- If the stored conditions have been overwritten, we abort.
- If we have overwritten the end of the queue, we abort.
- If we have written into the "danger zone" of the queue, we reset the tallies.

The scu/tra/queue management is explained as follows: To avoid writing over conditions in the queue, there is a variable scu/checks for each real processor that holds the address of the scu block from which conditions were last stored for this processor. Lub/addr (upper) is the smallest of these addresses and we try to never write past it; we consider the information ahead of this address no longer important. The only way there can be more than one scu/tra block pair per processor is if the tra instruction faults after the scu instruction is executed, or if there is a fault(s) in the fiim. The queue is set up with the following pointers, as shown below:

start/q, b/trsh, q/trsh, end/q, lub/addr

Every time the scu/tal is incremented, the tra/tal is also incremented. If we fault during the scu instruction, we get a system trouble fault and abort the system; so we do not have to worry about unmatched scu/tra blocks and we know that the information stored in the tra block is always stored after (i.e., more recently) than in the scu block.

If we did not have to worry about updating the tally words, the tra/tal would always point to the tra block immediately following the scu block that scu/tal pointed to. However, since other processors can be using the tally words while one processor is updating them in the fiim, we might encounter the situation of having a processor do an scu scu/tal, ad just before the updating of the scu and tra/tals (which are updated simultaneously), so that the tra tra/tal, ad instruction executed by the processor causes a transfer to the top of the queue while its scu conditions have been stored at the bottom of the queue. However, the tra/tal address is still circularly greater than that of the scu/tal so we use the address in tra/tal to check if we have over-written any important information.

We define the lub to be the lowest (number) of the addresses in the scu/checks and as the tallies are updated to the top of the queue and then incremented, we want to stop before this address in the lub. To help insure that we do not store past this address, we define b/trsh to be b/trsh/upd words before (circularly) the lub and try to stop at this address. We use the address in tra/tal to check for b/trsh.

If we find that the tra/tal address is within b/trsh/upd words of the lub, we abort. In the same manner, we attempt to avoid writing past the end of the queue by defining a q/trsh to be a fixed distance before the end of the queue and we update the tallies whenever the address in tra/tal is greater than this value in q/trsh.

Below are the six possible arrangements of the lub, b/trsh, and tra/tal:

lub	lub	b/trsh
b/trsh	tra	lub
tra	b/trsh	tra
1	2	3
b/trsh	tra	tra
tra	b/trsh	lub
lub	lub	b/trsh
4	5	6

Conditions 1, 4 and 6 should cause an abort.

Fiim--The last instruction is the tra block, a transfer indirect through an its pair, causes us to enter append mode. As its name implies, fiim is pure, i.e., it can be executed simultaneously by several processors. The outline below describes the operation of fiim.

• • •

A. INITIALIZE

1. The first ten instructions in fiim are inhibited. As soon as fiim is entered pr.rsdb and pr.lcdb are set to correctly reference the real system data base (rsdb) and low core data base (lcdb) respectively.
2. Then rsdb|inhib/scu/mask is used to set the memory controller mask (smcm) in the scu with lower order memory. Once this is done it is no longer necessary for the instructions to be inhibited.
3. Next, the processor switches are read to determine which real processor is executing the fiim. With this information we can load br.rpdb with a sdw number that corresponds to an unique segment for each real processor known as the real processor data base (rpdb). While executing the code that accomplishes this we set x7 to the processor number. There is an array, temp, with a word for every real processor. Thus we can use "temp, 7" as a temporary storage location and the code is still pure.

4. As soon as the rpdb has been determined the clock of scu0 is read and the value stored in rpdb vmm entry time.
5. Next x5 is loaded with lcdb|scu/tal. The conditions for the real processor were stored at least 40 words before this value, depending on how many processors have since executed a fault/interrupt vector pair.

B. FIND A VALID scu BLOCK FOR THIS PROCESSOR

In most cases there will only be one set (values stored in the scu and tra blocks together) of conditions stored for the real processor executing fiim (see the above discussion of how the scu/tra queue works). However, there may be two or more sets of conditions stored, and we want to find the first, or earliest, stored set in order to save the conditions with which fiim was entered. Since x4 contains the address of the tra block, we are searching only for the earliest (oldest) scu block for this processor.

To best utilize the queue we want scu/checks for this processor to be set with the address of the most recent scu block found for this processor, so we use rpdb/first-time ind as a flag; when it is nonzero we have not yet found a valid scu block for this processor; when it equals zero we have found a valid scu block and we have saved its address in rpdb|saved/tally/addr.

1. Find an scu Block for this Processor

As stated above, $x5$ is set equal to $lcdb | scu/tal$ so $x5$ either equals a valid scu block address or it is $> q/trsh$.

We search backwards through the queue looking for a scu block stored by this processor. Word 3 of the scu data contains the processor number in bits 28-29. If this is equal to $x7$ we check to see if we have reached the address of the scu block used for this processor the last time it was in fiim. This address is kept in $rsdb | scu/checks, 6$. If we have not reached it yet we continue backing up the queue; if we reach $rsdb | scu/checks, 7$ we decide that there was no block of conditions stored for this processor and we abort, abort/6.

2. See if Address for this scu Block is Valid

$x5$ is the address of a scu block stored by this processor. We check bit 18 or the third scu word to see if the conditions have already been stored. Unless we are in the "possible garbage area" ($x5 > q/trsh$) we abort if the conditions are already stored, abort/8. This bit (18) is set near the end of the fiim and is reset when new scu-conditions are stored in this block via the scu instruction in the fault/interrupt vector pair.

If the conditions have not yet been picked up and stored, we proceed under the assumption that either (i) the scu conditions are valid, (ii) they were stored as a result of a fault in the fiim and we will continue looking for the earliest block for this processor, or, (iii) something is wrong with the queue and after discovering this later in the fiim we will abort.

As may-be-valid-conditions we set up pr.vmdb and pr.vpdb to reference the virtual machine data base (vmdb) and virtual processor data base (vpdb) respectively. Then we store the second word of the scu conditions into rpdb/scu/fault/word so that we can pick it up before leaving fiim. We cannot wait until then to look for this second word because x5 will no longer reference the scu block and pr5 will not be valid pointer to stored conditions in the case of an interrupt in the VMM.

C. STORE CONDITIONS

1. Determine Type of Conditions -- Fault or Interrupt

From word 2 of the scu block, lcdb/scu/fault/data/word, 5, we can check bit 35 to see if we had an interrupt or a fault; then from the real/virtual indicator, bit 32 of the fifth word of the scu conditions, we can discover if this fault/interrupt occurred in the VMM or the VM.

If there was a fault in the fiim we return to look/for/scu to find another scu block for this processor.

If there was a fault in a VM we set bit 0 of rpdb|ind/word to show that fiim was entered from a VM by a fault.

If there was an interrupt we set bit 6 in rpdb|ind|word.

If the interrupt was in the VM we zero bit 0 of rpdb|ind/word to show that fiim was entered from a VM by an interrupt.

2. Set up pr5

We pick up conditions from lcdb/0, 5, where x5 is the address of an scu block, and store them in 5/0, where pointer register 5 -pr5- has been set up to reference the first word of a 48 word block of machine conditions. pr5 is set as follows:

Fault/Interrupt in the VMM--For a fault in non-fiim VMM, pr5 points to rpdb|vmm/cond/start, 3. vmm/cond/start is the first address of a list of machine conditions blocks, and x3 determines which block we will store into.

For an interrupt in the dispatch segment, no conditions are stored and pr5 has no meaning in this instance.

Fault/Interrupt in the VM--For a fault or an interrupt, pr5 references vpdb|vp/mach/conditions.

3. Store the Conditions

Pointer and lengths are stored at all times.

For the mlr instructions we have to make sure that we work with character addresses; in the case of desc9a these character addresses are four times the word addresses. We store the scu conditions and the value of the clock that we read on entering fiim and then saved in rpdb|vmm/entry/time.

From x4, the word address of the tra block, we can determine if we stored conditions in a type1 of a type2 block. From each block we pick up and store the pregs, regs, and the dsbr.

A 'scpr' with a tag of 01 stores the fault register in 5/mc/fault/reg.

4. After all conditions have been stored, the stored bit is set in the scu conditions in the queue.

D. QUEUE UPDATE

In the queue updating code we update scu/checks, the lub, b/trsh, etc. if necessary while checking for possible error conditions.

On reaching queue-update we check the stored bit of the lub. If it is one we proceed, but if it 0 the lub has been overwritten and we abort since we can no longer be confident that the conditions we picked up from the scu/tra queue are valid. (abort/18)

While updating the queue we lock the code with rsdb|fiim/q/lock.

1. Update scu/checks

We store the address of the most recent scu block found for this processor (this address has been saved in rpdb/saved/tally/addr) in scu/checks. 7.

2. Update lub, b/trsh, etc. if necessary

If this processor owns the lub we have to search all the scu/checks of the active processors to find the lowest address. We put this address into the lub, i.e., into rsdb|lub/addr (upper half) and calculate a new rsdb|b/trsh by circularly subtracting rsdb|b/trsh/upd words. If the lub is less than b/trsh/upd words from the start of the queue (rsdb/start/q) we wrap around to the bottom of the valid queue, above rsdb|q/trsh, and continue moving backwards until we have gone b/trsh/upd words.

3. Check Queue Position

There are six possible queue positions with respect to the lub, tra/tal, and b/trsh. We check for the three acceptable positions and abort for the other three, wait/abort. See the information on the scu/tra queue above for a discussion of which positions should cause aborts.

E. TRACE

If `rsdb|trace/flag = 0` we do not perform a trace. If it is not equal to 0 we enter trace by one of two entries:

Entry point 1 (`trace/1`) is used when we wish to save the trace code, words 2, 5, and 7 of `vp/mach/conditions`, `rpdb|vm/vp/no`, `ind/word`, `vpdb|vp/state`, and the real proc. number.

Entry point 2 (`trace/2`) is used when we wish to save the eight words we have stored into `rpdb|trace/conditions`.

`Trace/1` and `trace/2` are both entered with a `tsx6`, `x5`, `x7`, and `pr6` are changed for both entry points; `rpdb|trace/conditions` are changed if we enter at `trace/1`.

F. LEAVE flim

The `Q` register is loaded with the fault/interrupt number, right justified.

`Rpdb/interrupt/flag` is used to tell whether it was an interrupt or a fault that brought us to flim. If `interrupt/flag = 0` it was a fault and we transfer to `f-proc` through an its pair. If `interrupt/flag` does not equal zero we transfer to `i-proc` through an its pair to handle the interrupt. In both instances we enter the second word of the particular segment since the first word is an error trap.

• • •

Functional Description of I-proc

I-proc is the VMM module which handles the processing of all VMM and VM interrupts. I-proc is invoked by the fault/interrupt interceptor module after the occurrence of an interrupt. Upon entry to this module the state of the processor has been saved for those interrupts which occurred while the processor was in virtual mode. If the interrupt happened in real mode (i.e., while in the VMM) then the state of the processor will not have been saved. The reason the processor state is not preserved is that while in real mode the processor is inhibited from receiving interrupts except during one instance while in the VMM dispatching module. If an interrupt occurs at this point, it is not necessary to save the processor state. Interrupts cannot occur at any other instance because the processor is inhibited through the use of the inhibit bit and the SCU masks.

The processing performed by I-proc and its related routines entails a simulation of the functions performed by the I/O multiplexor (iom) and the system controller (scu). For this simulation I-proc must appropriately update the virtual address space corresponding to the virtual machine associated with the interrupt being processed. This updating consists of modifying the virtual mailboxes, interrupt multiplex words, and status words.

Usage Information for I-proc--This module relies completely on the data-bases constructed for the real and virtual environments. When the interrupt processor is invoked by fiim, it is expected that the interrupt type (int-level) will be contained in the lower portion of the Q register. Upon exit from I-proc, the appropriate virtual addresses and VMM data bases will have been updated to reflect the completion of the interrupt processing.

Resource Objectives of I-proc--The objective of the interrupt handler is to efficiently simulate for a virtual machine the operation of the peripheral subsystems in regard to the termination operations performed in processing on I/O request.

Design Overview of I-proc--At this level of the interrupt processor the current time of day is read from the scu with memory address 0. This is used to meter the time spent processing interrupts. Then according to the type of interrupt being processed as determined by the interrupt level number, the module corresponding to I/O system first initiating the interrupt will be invoked. All processing of the interrupt will be performed at these lower levels with the aid of a set of common subroutines.

Functional Description of iom/int--The processing of all interrupts from an iom are handled by iom/int. The amount or extent to which the interrupt is processed and simulated for the corresponding virtual machine is dependent upon the interrupt type.

At this level, the interrupt multiplex word, imw, corresponding to the interrupt received is examined to determine which if any channels require interrupt processing. Each channel interrupt is processed independently. If the interrupt is from the overhead channel six then processing will be performed by the psia-proc module. All other interrupts from overhead channels will cause a VMM abort.

The processing of payload channel interrupts is handled separately. Non-terminate interrupts will be handled by the module non/term/int. If the payload interrupt is the result of an interrupt from a 355, then the

module DN355 will be invoked. The only interrupt not covered by the above two cases is the terminate interrupt. The terminate interrupt processing consists of determining the real-to-virtual map, updating the virtual mailbox entries, adjusting and storing the associated status word, and setting the proper virtualimw bit. After this processing has been performed the data storage for this request is released and the processing of the next I/O request for the real channel or any of its crossbars is initiated by iom-proc.

Usage Information for iom/int--This submodule is called from within the interrupt processor, I-proc, to process any iom related interrupts. Upon entry the number of the real iom causing the interrupt is known and the type of interrupt is provided by the imw level. The contents of the imw services to drive this module such that all channel interrupts are processed. Upon termination of this routine the interrupt has been simulated in virtual space and the appropriate interrupt cell in the virtual machine's scu database has been set. Also, a connect has been performed for the next queued I/O entry for each channel processed.

Functional Description of non/term/int--This routine is used to process interrupts which are not expected by the VMM (i.e., non-terminate interrupts) with the exception of channel six interrupts. Since they are not expected, the virtual mappings are not determined as in the terminate case. Instead, the mapping is calculated from a set of mapping constructed at VM start-up time. Once the virtual map is determined, the VM's corresponding imw and SCW interrupt cells are set to complete the processing of this interrupt.

Known Limitations of non/term/int--This module assumes only dedicated peripherals are configured to a VM. In addition, only special interrupts and interrupts from a 355 will be processed. If a marker or initiate interrupt is received it will result in a VMM abort.

Functional Description of real/virtual/map--This module transforms the mapping of the real device currently being processed into a mapping of the associated virtual device. The virtual machine, iom, channel, and device numbers are determined as well as descriptors describing the VM's data base within rsdb and the VM's address space.

Usage Information for real/virtual/map--The input variable vir-map (qu) is used to create the variables vir-iom(x1), vir-ch(x3), vmpr(pr6) and vmdbtr(pr7). After the module is finished executing, control will be returned to the x6 within the current segment.

Functional Description of update/virtual/scw--This module is responsible for updating the virtual status control word(scw) corresponding to the virtual channel whose interrupt is currently being processed. This updating shall be identical to that performed by the iom-B on a status service. The iom-B allows three types of status queues - a list, a circular queue with four entries and a circular queue with 16 entries. The type of queue as well as the next entry in the queue is specified by the scw. A tally field is also provided to determine the length of a status list when the list option is employed.

Usage Information for update/virtual/scw--In order to process the virtual scw, this module must be called with the following variables:

1) vir/iom

2) vir/ch

3) rpdb

4) vmpr

5) vmdbtr

The above variables will be preserved during the module's execution. In addition, the virtual scw corresponding to the interrupt to be processed will be updated and vir/scw/address will be the virtual address used to store the status.

Design Overview of update/virtual/scw--The function of update-virtual-scw is to perform the same type of servicing for the virtual scw as performed by the iom for a real scw. This includes updating the scw pointer and decrementing the scw tally. The exact procedure for this scw modification is presented in detail in the iom EPS-1.

In addition to the update function, the routine also tests the store address for the next status pair to insure that the status will be stored within the proper virtual machine's address space. This address is used by update/virtual/status to store the virtual status.

Functional Description of update/virtual/status--The function of update/virtual/status is to perform the updating of the virtual mailbox lpw and lpwe and the virtual status pair. The virtual mailbox scw is updated by the module update/virtual/scw.

Both the lpw and lpwe are modified by the iom in the process of executing a channel program. When an interrupt is received from this channel, the state of these two mailbox words may reflect the presence of the VMM. Therefore, the address fields and lpw state bits (AE, relative, restricted) must be virtualized.

The status pair is similar to the lpw and lpwe in that it must also be virtualized. The first word of the status pair contains an address extension field which must be corrected to reflect the virtual address extension. The dcw residue word (the second word of the status pair) must also be virtualized to reflect a possible VM address space offset from a 256K memory boundary.

Usage Information for update/virtual/status--The following variables must be provided to update/virtual/scw upon entry:

- | | |
|--------------|-----------------|
| 1) lcdb | 6) vir/ch |
| 2) rpdb | 7) vmpr |
| 3) real/iom | 8) vmdbtr |
| 4) real/chan | 9) vir/scw/addr |
| 5) vir/iom | |

These variables will not be modified during the execution of this module. However, this module does update the virtual channel's lpw, lpwe and status pair within the VM's address space.

Resource Objectives of update/virtual/status--The updating of the lpw, lpwe, and status pair adds greatly to the overhead involved with processing an interrupt. This is especially true when the virtual machine ignores the virtualization in most cases.

Known Limitations of update/virtual/status--In order to keep VMM overhead at a minimum, a complete virtualization of the VM's channel mailbox is not always performed. Only in the event of status pair with non-zero major or minor status fields will the VM's lpw and lpwe be updated. It is assumed that the lpw and lpwe will not be examined when the major/minor status fields are zero.

Functional Description of set/vir/imw--One of the functions of set/vir/imw is to simulate the scu in setting its interrupt cells. Within the system control unit (scu) there exist 32 interrupt cells. Each of these cells corresponds to one of the 32 interrupt types. The proper cell is set when the scu receives a set interrupt cell command from one of the subsystems configured to the scu. These cells together with the scu's interrupt can be delivered to a configured processor. This same function is simulated by set/vir/imw by setting the corresponding bit in the virtual machine's data base word representing the scu interrupt cell.

Another of the functions performed by set/vir/imw is to set the proper imw word in the virtual machine's address space. When the iom sends the scu a set interrupt cell command, it also requests the scu to set a bit in the imw word corresponding to the interrupt cell just set. The bit set within the imw word represents the channel on the iom which initiated the interrupt. This iom/scu function is simulated by setting the proper bit in the virtual machine's imw.

The final task performed by the set/vir/imw is to construct a trace entry. When the VMM system trace indicator is true, a trace entry consisting of the real status pair, the real channel index, and the interrupt type is built and the system trace module is called.

Usage Information for set/vir/imw--This module requires the variables vmpr (pr6) and vmdbtr (pr7) be specified in the described hardware registers.

Functional Description of dn355/int--The purpose of this module is to process interrupts from 355s which are connected through the modified DIA to the iom. For VMM step 1.1 development only, dedicated 355s will be allowed. A dedicated 355 is one which is only configured virtually to one virtual machine.

Under the dedicated 355, the 355 mailbox directly will be accessible to the 355 and may be directly updated without VMM intervention. Therefore, 355 interrupt processing will not require a simulate of the operations performed by the 355. However, the scu functions must still be simulated for the corresponding virtual machine. This involves the setting of the appropriate interrupt cell in the VM's virtual scu's data base and the updating of the corresponding virtual imw within the VM's address space.

Usage Information for dn355/int--This module is called from the iom interrupt handler when an interrupt from a 355 is detected. When called the interrupt is processed as explained earlier. Upon entry to the module the 355 channel index is known (x5) as well as the address of the channel's device descriptor table (a1). Most of the processing of the 355 interrupt is performed by the two modules: real/virtual/map and set/vir/imw.

Design Overview of dn355/int--The functionality of this module is simplified by the use of the two modules: real/virtual/map and set/vir/imw. Upon entry to dn355/int, the real to virtual map for the dedicated 355 is obtained from the first entry in the channel's device descriptor table. The real/virtual/map routine is then called to determine the virtual machine number, virtual iom number, and the virtual channel index. This later module also sets up the pointer registers for the VM's data base and virtual address space. Upon return the module set/vir/imw is called to set the VM's corresponding imw and to set the proper interrupt cell in the virtual scu data base within the VMM. At this point the processing of the 355 interrupt is complete and control is returned to iom/int to process the next channel interrupt.

Functional Description of vmm/int--This routine processes the VMM software interrupts. The four VMM software interrupts, one for each of four possible real processors on a VMM system, are assigned to interrupt cells 3, 8, 9, and 10 respectively. These software interrupts are set by a smic instruction in dispatch when there exists an outstanding connect fault or interrupt for the vm/vp currently executing in that real processor. The use of these interrupts allow the virtual processor to be interrupted when the vp reaches an interruptable state.

Design Overview of vmm/int--The processing of a VMM software interrupt simply involves the resetting of the flag: proc/spec/int/flag. This is used to indicate that the VMM software interrupt cell for this real processor is not set.

Functional Description of f-proc

The purpose of the fault processor is to determine the reason for a hardware fault and process it accordingly. For metering purposes, the fault processor will keep statistics concerning the time taken to process faults. There are two major categories of faults distinguished by the fault processor:

- 1) VMM faults--These are faults that occur within the VMM and are, therefore, up to the VMM to process for itself.
- 2) VM faults--These are faults that occur while a virtual processor "has control" of a real processor. These faults range in processing by the VMM from completely to not at all.

There are 36 different "hardware" faults for each of the above mentioned categories, making a total of 72 individual fault processors. These individual processors are described in lower levels of documentation.

Resource Objective of f-proc--The fault processor should be time optimized as much as possible to the more common functions. The least common functions need not be so optimized.

Design Overview of f-proc--The time the fault processor was entered shall be saved for metering purposes. Metering shall be accomplished at the end of fault processing and is dependent of the metering routine defined in a lower level of documentation. The input parameter, the fault number causing fault processor entry, together with the variable holding the count of VMM faults not yet processed determines the fault processing to be

attempted. If there is a VMM fault outstanding, then the input parameter will be assumed to give the number of the last fault (not yet processed) that occurred within the VMM. The fault processor will always handle faults occurring within the VMM before it handles those occurring within a VM.

Once the fault processor determines where the fault occurred, it determines what fault handler to select (depending on the input parameter). The appropriate fault handler is then invoked. Metering is done on return from the particular fault handler.

Temporarily all faults occurring within the VMM will cause the system to abort.

Variables for f-proc--All variables are defined in their respective include files. All variables referenced at this level of abstraction are in the rpdb include file.

Known Limitations of f-proc--No provision for the handling of faults occurring within the VMM has yet been made, except that the system will abort. Metering measurements are slightly biased toward the low side due to the time of execution of the metering instructions themselves. Also, when getting to processing VMM faults, the entry time to the fault processor may be overwritten before a measurement is actually made (thus biasing to the low side).

Functional Description of vp/ivr

If the illegal slave bit in the scu conditions is not on, then the fault is returned as is. The virtual fault register is updated appropriately.

If the illegal slave bit is on, then the modes are checked to see whether the instruction was executed in:

- 1) 6100, absolute, master modes
- 2) 6100, append. master and privileged (PPR.P=1)
- 3) 6000 and master modes

If the instruction fault was executed in 6000 and slave modes, then a table lookup on the faulting op code is performed. If the op code is in the table, then x3 will contain the address plus one of the locations associated with the "faulted on" op code. The VMM special command processor (VMM/cwd/proc) will handle the next step in instruction simulation. If the op code is not in the table or the instruction was not attempted in any of the above mentioned modes, then an illegal-in-slave fault must be returned. For MULTICS, illegal-in-slave processing requires only that the virtual fault register be updated. For GCOS, illegal-in-slave processing requires that the fault number be changed to a command fault (in the sw conditions) as well as that the virtual fault register be updated.

Functional Description of vp/acv

The access-violation fault is one of the 6100 faults that may need to be returned to a 6000 virtual processor (under a different fault name, of course). The only access violation type that will be returned to a 6000 vp is

the out-of-segment-bounds fault. This fault will occur (on a 6000 vp) only if memory is mapped via a segment descriptor word. If an out-of-segment-bounds fault is detected for 6000 vp, then a store fault (non-existent-address) is simulated for the vp. An access violation occurring within a 6100 vp will be given back to the vp as is.

Design Overview of vp/acv--If the access violation occurs while a 6100 vp has control of the processor (vmdls/vm/type/0), then the fault will be returned to the vp as is. If the access violation (acv) occurs while a 6000 vp is in control, then two cases are relevant:

1. If the acv is an out-of-segment-bounds, then the 6000's memory is assumed to be bounded by a segment descriptor word. A store fault (non-existent address is returned).
2. If the acv is not an out-of-segment-bounds, then the VMM is expected to have set up the decor incorrectly and the VM is aborted.

If a store fault is to be returned:

1. The out-of-segment-bounds bit in the scu conditions is zeroed out.
2. The non-existent address bit in the scu condition is set on.
3. The fault field (containing access violation number) is replaced by the store fault number.

4. The virtual fault register's non-existent address bit is set on.

Known Limitations of vp/acv--If GCOS is guaranteed to no longer run with memory bounded by an sdw, then the store fault simulation can be eliminated (11 instructions).

Functional Description of absa/routine

The absa routine is an address development package used in coordination with the various instruction simulation modules within the VMM. It develops an 18 bit effective address or 24 bit virtual final address according to the instruction being simulated and the mode under which the instruction would have been executed.

Usage Information for absa/routine--This address development routine is invoked via a call to one of the three entry points as follows:

tra 1cdb		vm/addr/1/entry
tra 1cdb		vm/addr/2/entry
tra 1cdb		vm/addr/ea/entry

A 24 bit final virtual address is constructed via a call to vm/addr/1/entry with no index register arguments. The entry point vm/addr/2/entry is used to develop a 24 bit final virtual address and it must be called with absa/inst and absa/ic/ir in the A and Q registers respectively. The final entry point vm/addr/ea/entry develops an 18 bit virtual effective address and requires no index register arguments.

Upon return to the calling module the appropriate address will be right justified in the upper 24 bits of the 4 register. The contents of all other index register may have been destroyed.

Resource Objectives of absa/routine--Execution efficiency should be the primary objective while memory usage shall be secondary.

Design Overview of absa/routine--The purpose of absa is to simulate the address development for a virtual instruction's operand. This simulation is performed by creating the exact environment under which the instruction would have been executed in virtual mode. This includes restoring all virtual index registers, pointer registers, dsbr, ic and the pertinent control unit information. The processor mode is described by the state of the following indicators as follows:

- Zero: master/slave. 1 \Rightarrow master
- Overflow: absolute/append. 1 \Rightarrow absolute
- Exponent underflow: 6000/6100, 1 \Rightarrow 6000
- Negative: 18 bit effective/24 bit final, 1 \Rightarrow 18 bit effective address

Once the virtual state has been restored as described above, the virtual instruction is executed with the instruction's op/code replaced by the absa op/code. This absa instruction performs the appropriate address development under the control of the aforesaid indicators. After the construction of the virtual address, control is returned to the caller.

The packaging of the absa routine is unlike most VMM routines. Initially, the routine is entered in append mode in segment lcdb. In lcdb the index registers and indicators are set up and then control is passed to the rpdb in absolute mode. At this point the virtual dsbr and pointer registers are restored and then a restore control unit is performed to reconstruct the remaining virtual conditions such as psr/ppr, and ic. The restoring of the control unit results in the execution of an xed pair containing the modified absa instruction and a tra. The absa develops the virtual address while the tra causes a return to the VMM in rpdb. The VMM's registers are then restored before returning to calling module.

FUNCTIONAL DESCRIPTION OF DISPATCH

The composite of modules comprising dispatch is responsible for the allocation of a real processor to a specified virtual processor/virtual machine. The control routine for the dispatching process is invoked after the VMM has completed the processing of the exception (interrupt/fault) which engaged the VMM. At this point of entry, the virtual processor which was in execution at the time of the exception is now ready to be assigned a real processor.

However, before this processor or any processor is dispatched, the VMM will accept any outstanding interrupts (the interrupt set directly by the VMM is the only exception). The dispatch control routine is the only instance within the VMM where an interrupt is permitted. If an interrupt is present, control will be passed to the fault/interrupt intercept module via the corresponding interrupt vector and the interrupt will eventually be processed by I-proc. In the event that no interrupt is outstanding, the VMM will again be inhibited from receiving interrupts until a virtual processor is placed in execution.

After permitting all interrupts to be processed, the dispatch control routine will determine the dispatch state of the virtual machine/virtual processor receiving the initial exception. If this virtual processor's execution can be suspended, the dispatch module NEED will be invoked to determine the next virtual machine/virtual processor to be dispatched. The conditions upon which a virtual processor's execution may be suspended are:

1. The virtual processor is currently executing a "dis" instruction.
2. The virtual processor was interrupted and
 - a) the virtual processor is a control processor (i.e., the virtual processor can receive interrupts) and its interrupt masks are set to allow interrupts, or
 - b) the virtual processor is a non-control processor and it was executing in slave mode when the interrupt occurred.

Design Overview of dispatch

Of the two functions performed by dispatcher/interrupt recognition and dispatching/interrupt recognition is handled first and dispatching is performed later by subroutines of dispatcher. Entry to dispatch occurs after the processing of a fault/interrupt by F-proc/i-proc. Upon entry, the scu masks are set to allow any outstanding interrupts. The occurrence of an interrupt results in control being delivered to film via the interrupt vectors. After the processing of the interrupt, control is eventually passed to dispatch where a test for additional interrupts will be made. If no interrupts

are awaiting processing, dispatcher determines to which eligible virtual processor it will place in execution. The criteria for dispatching is as follows:

- A. If the VMM was entered via a valid VM fault (i.e., not a fault occurring as a result of the execution of a privileged VMM instruction), the real processor will continue to execute on the current virtual processor.
- B. If entry to the VMM resulted from the attempted execution of a privileged VMM instruction, then the processor will be dispatched to the same vp. This is accomplished by the VMM routine `vm/spec/fault`.

The `dis` instruction is an exception to the above dispatching criterion and control is not returned to the same vp. Instead the module `NEED` is called to find an eligible vp for dispatching.

Functional Description of `vp/int`

This module will cause the return of a vp via a simulated external interrupt if there exists a pending external interrupt--any interrupt of a connect fault. If no external interrupt is pending, then the virtual processor will resume execution at the point where it was initially interrupted.

Usage Information for `vp/int`--When control is given to this module, `x4` must contain dispatch entry. Upon exit from this module the vp will be placed in execution.

Design Overview of vp/int--This module along with its submodules is used to return to a virtual processor whose execution has halted as a result of an interrupt. As such this module will place the processor in virtual mode via a simulated connect fault, simulated interrupt, or return to the next instruction following the real interrupt. The exact method of return is dependent upon the existence of a connect fault or interrupt which should be delivered to this virtual processor. If either condition exists, then return would be via the appropriate vector. In the event that both exist, the connect fault will have higher priority.

If there still exists pending interrupts after determining the method of return, these will be queued. In order to deliver these interrupts, the processor's VMM interrupt call will be set in the system control unit. This forces the virtual processor to be again interrupted when it enters non-inhibited code.

Functional Description of vector/simulation

Vector/simulation, as its name implies, simulates the instructions in an interrupt or fault vector pair. When control is to be returned to a vp by means of a fault or interrupt this routine is invoked. For non-transfer type instructions in the vector pair, an actual simulation of the instruction will be performed, whereas a transfer instruction will actually be executed in the return to the vp.

Usage Information for vector/simulation--In addition to the normal register connections, vector/simulation must be called with the address of the vector pair relative to the vm base in x3 (vector-base).

Resource Objective of vector/simulation--It is not the intent of this module to simulate the operation of every 6100 instruction. This in itself would be a major undertaking and although it is necessary for the operation of a functional VMM, it is not practical. Therefore, only the instructions currently used by MULTICS and GCOS in their vectors will be simulated. At present this set of instructions includes tsxn, tra, rcu, stcl, lda, ldxn, slxn and nop. The addition of other instructions would be no more complex than the instruction's simulation.

Design Overview of vector/simulation--Vector/simulation is used to simulate the pair of instructions found in a vp's interrupt or fault vector. As mentioned in the resource objectives, only specific instructions are expected in these pair and only these will be simulated. A linear table search is used to determine if the instructions in the current search is used to determine if the instructions in the current vector pair are one of these specific instructions. At the same time, the corresponding simulation routine will be located.

If the instruction to be simulated is a transfer type instruction, the instruction is implanted into the even instruction portion of the vp's control unit block. The appropriate mode indicators are then set to simulate the state of a processor while executing a vector pair. These two operations will result in the proper execution of the transfer when the control unit is restored upon the vp's dispatch.

For the non-transfer instructions, the final address of the instruction's operand is developed. For MULTICS this entails invoking the absa routine, while a GCOS vp requires a simple simulation for most instruction tags.

After the final address is developed, the instruction itself is simulated. After this simulation control is returned to dispatch.

Known Limitations of vector/simulation--Faults within a multi-word instruction in a vp without a transfer in the corresponding vector will result in a VM abort. The same is true for an interrupt within MULTICS.

Functional Description of vm/int/test

Vm/int/test is involved during the VMM's dispatching algorithm to determine if there exists an outstanding virtual interrupt which the vp to be dispatched can process.

Usage Information for vm/int/test--Entry to vm/int/test is via a TSX6 vm/int/test and upon return int/type will describe the interrupt type via which the dispatch to the vp will be made.

Resource Objectives of vm/int/test--Under current VMM development, it is envisioned that a vp will not be receiving interrupts from more than one scu. *This is true of the current releases of both MULTICS and GCOS.* As such the VMM will only check the control scu for outstanding interrupts.

Design Overview of vm/int/test--In testing for outstanding virtual interrupts, vm/int/test compares the virtual interrupt mask with the interrupt cells for the vp's virtual control scu. If no match is found between corresponding bits of the mask and interrupt cells then control is returned to the calling module. However, if a match is found and the vp cannot currently receive

an interrupt, then the interrupt cell corresponding to the current real processor is set in the scu with the memory bank containing real address zero. This will cause this and only this real processor to be interrupted in virtual mode as soon as non-inhibited code is executed.

In the event of a match and the vp can be dispatched with an interrupt then the entry for the first such match will be removed from the virtual scu's interrupt cell. If more than one match exists, the interrupt cell corresponding to this real processor would be set in the scu with memory address zero. Control will finally be returned to the caller with the type of interrupt that the vp will be dispatched with.

Known Limitations of vm/int/test--Entry and exit to vm/int/test should be gated.

SECTION 3

DESIGN APPROACH FOR VMM EXTENSION

The various functions performed by VMMs can be segregated into two broad categories: those which are frequently used and those which are infrequently used. Some examples of these, as a function of their frequency, are the following:

- **Frequently used functions**
 - Real I/O initiation
 - Fault and interrupt processing
 - Dispatching of processors
 - Simulation of faults/interrupts in virtual machines
- **Infrequently used functions**
 - Virtual machine definition
 - Virtual machine start-up
 - Receipt and response to virtual machine operator commands

All of these functions are the same or similar to functions which are normally performed by an operating system. This suggests that the functions with low use frequency needed in a VMM might be supplied by an operating system executing in one of the virtual machines, the Service Machine (SM), being supported by the VMM. The primary advantage of this approach is that the code to support such functions need not be redundantly created and maintained within the VMM. If, in addition, the functions supported by the operating system in the SM exist in the form of user (slave) programs, some important secondary advantages accrue:

- The overall VMM functionality is (at a point in time) richer because of the relative ease of creating user programs.
- The incremental resources committed exclusively to VMM are relatively small because the operating system in the SM executes ordinary slave programs on behalf of other users in addition to providing service to the VMM.
- The currentness of the VMM with respect to new devices is maintained with relative ease by simply using correspondingly new versions for the operating system in the SM.

The service machine concept could be used in providing enhanced functionality in the VMM. In fact, the design effort on the extended functionality has proceeded sufficiently far to confirm the feasibility of the service machine approach.

GENERAL AREAS OF FUNCTIONAL NEED TO BE EXAMINED

We wish to examine techniques for extending the VMM functionality to include support for:

- Shared or dedicated unit record peripherals
- Shared front end processors
- Simulated system consoles
- General user interfaces

The service machine concept can be applied effectively when the overhead incurred with this approach is not significantly detrimental to system performance. Each of the above designated functional areas has some aspects which are invoked relatively rarely (e. g., definitional) and others which are invoked frequently (operational). Thus, we would expect a design for these functions to support some aspects via the service machine technique and others via direct extension to the base VMM and hardware.

All of these functional areas relate to the handling of I/O for virtual machines.

GENERAL I/O CONSIDERATIONS FOR VIRTUAL MACHINES

We shall examine the handling of I/O by the VMM and address the subject of the treatment of shared and unit record peripherals in this section. These considerations apply to all the functional areas listed above.

I/O Device Specification

The reference to a device in virtual machine I/O needs to be mapped. The VMM has for each virtual machine a table of resources that can be consulted to determine the actual support being used for the virtual machine devices. Distinct forms that this support might assume are discussed in detail below in the section on virtual device support.

In the simplest form of device support, the VMM maps each virtual machine device into some real device of the same type. The VMM then merely substitutes references to the real device for references to the virtual device in the I/O operations.

I/O Program Analysis

The approach to mapping memory addresses and device references is actually more complex than described above. First, the mapped form of the I/O program, with the virtual addresses replaced by real addresses, cannot appear in the addressable memory of the virtual machine. If it did, other elements of the VM, such as the virtual processor, could reference the I/O program and read or alter absolute addresses. Therefore, the VMM must analyze the virtual machine I/O program and construct a translated I/O program in a work area private to the VMM.

Second, the translated I/O program may not be the same size as the virtual I/O program. For example, in a machine using a paging mechanism for memory mapping, a contiguous region of virtual memory need not map a contiguous region of real memory. Thus, I/O commands that involve data transfers crossing page boundaries might have to be split into multiple commands in a real I/O program. Again, the VMM must construct the real I/O program in a private work area.

A third complication arises in dynamically modifiable I/O programs. The central processor, the I/O program itself, or even some other simultaneously operating I/O program may attempt to alter the virtual machine I/O program during its operation. To reflect this functionality of a real machine fully would require a complex and large addition to the VMM and would introduce significant overhead processing when invoked. Third-generation VMMs in general reflect the conclusion that providing the full functionality is not required in a practical sense.

Any dynamic modification to an I/O program must be checked by the VMM to verify that the new form of the program will reference only resources assigned to the virtual machine. By not supporting dynamic modification of virtual machine I/O programs, it is considerably easier to ensure the integrity of the VMM and all virtual machines. Since a real I/O program is constructed in a work area private to the VMM, it cannot be modified by the virtual machine's processor or by any other I/O program of the virtual machine.

Caution must be observed when moving operating system software from a virtual machine to a real machine. Since dynamically modified I/O programs may behave differently on virtual and real machines, it is possible to develop software that operates correctly on a virtual machine but fails on a real system.

I/O Summary

In summary, basic third-generation virtual machine I/O operations are handled in the following sequence. The processor attempts execution of the instruction (cioc) to start an I/O processor executing an I/O program. This causes a trap to the VMM, which analyses the I/O program, maps memory addresses and device references, and creates a real I/O program in a work area private to the VMM. The VMM then queues the real I/O program for execution by the real I/O processor. When execution of the I/O program is completed, an interrupt to the central processor occurs. This causes control to return to the VMM, which analyses the case of the interrupt. If the interrupt signified I/O completion for some particular virtual machine, the VMM simulates the occurrence of an interrupt for

the virtual machine by manipulating the saved state information of the virtual machine. The VMM then makes a dispatching decision taking into consideration the interrupt event. Thus, the software on a virtual machine receives notification of I/O program completion precisely as it would on a real machine.

VIRTUAL DEVICE SUPPORT

Many applications for VMs depend on the ability of the VMM to provide an appropriate type of virtual device support. Figure 1 illustrates four different categories of support: dedicated, partitioned, mapped, and simulated. The mapping of device names and device data addresses, that is, addresses of data objects on devices, is discussed below for each of these support categories.

Dedicated Support

In the dedicated mode of support, the VMM maps a device of the virtual machine into an identical device of the real machine. No other virtual machines are allowed any form of access to the real resource. An example would be a disk drive assigned to a particular virtual machine. The particular device name may be different in virtual and real machines and is mapped by the VMM.

Partitioned Support

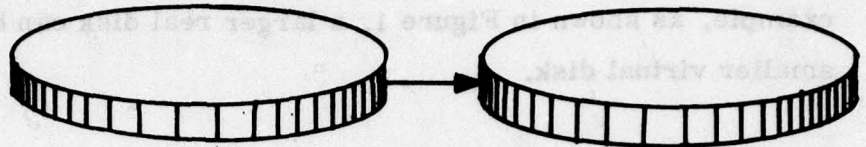
In partitioned support, the data addresses on the virtual device correspond directly to those on the assigned real device; unlike dedicated support,

SUPPORT

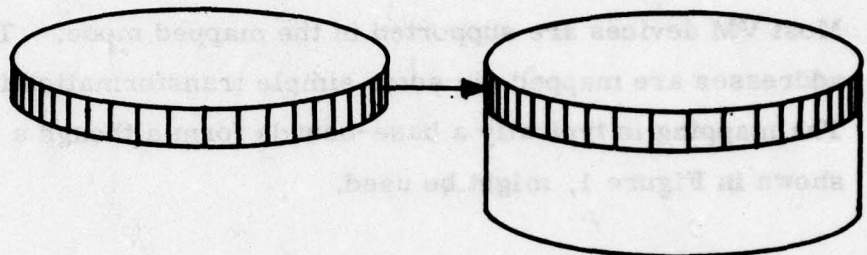
VIRTUAL DEVICE

REAL DEVICE

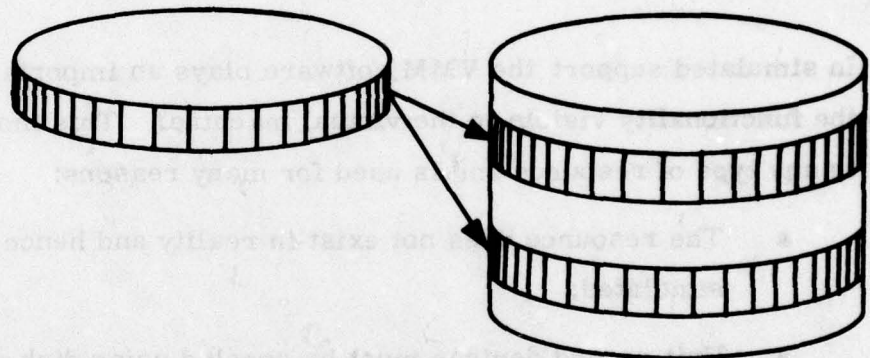
DEDICATED



PARTITIONED



MAPPED



SIMULATED

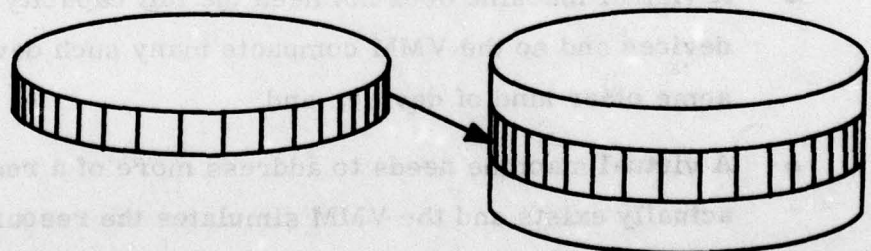


Figure 1. Modes of Virtual Device Support

however, the real device need not be identical to the virtual device. For example, as shown in Figure 1, a larger real disk can be used to support a smaller virtual disk.

Mapped Support

Most VM devices are supported in the mapped mode. This means that data addresses are mapped via some simple transformation into real addresses. The mapping is typically a base-bounds form although a multi-area form, as shown in Figure 1, might be used.

Simulated Support

In simulated support the VMM software plays an important role in creating the functionality visible to the virtual machine. This mode can be applied to any type of resource and is used for many reasons:

- The resource does not exist in reality and hence must be simulated;
- Unit record devices must be spooled using disk or tape;
- A virtual machine does not need the full capacity of its virtual devices and so the VMM compacts many such devices onto some other kind of device; and
- A virtual machine needs to address more of a resource than actually exists and the VMM simulates the resource using other resources.

VMM Involvement

For each of the modes of support for I/O equipment, the VMM has a different degree of involvement. However, the basic idea of having control pass to the VMM at the start of an I/O operation and the eventual interrupt reflection by the VMM back to the virtual machine remains unchanged in all modes of support in third-generation VMMs. The VMM support required when simulating a resource is greater than for other modes, and can be arbitrarily complex depending on the definition of the functionality of the resource.

SPECIAL CONSIDERATIONS FOR FRONT END PROCESSOR FUNCTIONAL EXTENSIONS

The support of the front-end processor (DN) could be accomplished in several ways some of which require hardware modification to the equipment (DN355, etc.).

The DNs internally use a line numbering scheme by which they identify the source or destination of messages. Clearly, these line numbers need to be related to the line identification used within an operating system in a VM.

The relationship between DN line numbers and operating system line numbers could be maintained within the VMM. In this situation a fairly heavy burden of processing would fall on the VMM for a highly interactive or transaction processing environment since each transmission in or out would require software I/O interpretation of line addresses. An alternate approach is to incorporate modifications to the DN hardware and software which make the

DN aware of the VMM. In this case, the DN can be used to manipulate the line addresses. This approach is in effect placing part of the VMM in the DN for mapping real to virtual resources. The DN needs to be aware of the identification of the VM for which it is performing service. This can be accomplished by a mechanism for the VMM in the main machine to talk to its counterpart in the DN. In effect, the two VMM parts would identify to each other, for each transmission, which VM was involved.

The issue of secondary storage also needs to be addressed in the context of front end processors. The DN and the main machine can share secondary storage equipment and exchange information via records stored on the equipment. Certainly, for VM integrity considerations, the DN must not be permitted global access to the shared secondary storage. Even if access were permitted to that part of the secondary storage corresponding to a particular virtual machine, j, the DN software would have to be designed to use the appropriate line identifications to avoid confusing the VM operating system.

The only currently known technique for supporting shared secondary storage between DN and associated VM is to incorporate a VMM internal to the DN hardware so that the DN software for a particular VM uses the same line number identification as the VM. Although we have been discussing only line number identification information, other characteristics of communications hardware also need proper virtualization. For example, different types of real terminals might be used instead of the type supported within the VM. In this case, the VMM within the DN hardware could simulate the device.

SPECIAL CONSIDERATIONS FOR SYSTEM CONSOLES

It would be possible to use any terminal equipment as a system console with proper VMM and Service Machine support; in particular, using terminals supported under TSS in a SM using GCOS.

The system console is conceptually all the buttons, switches, lights, and keyboard available to the operator of a real system. To that extent, all operator functions need to be supported via the virtual machine's operator console.

One approach to supporting the console is to define modes of terminal usage. One mode is that in which the terminal behaves like the keyboard and printer of a real console. The second mode is that in which the terminal emulates the lights, buttons, and switches. There is also a third mode of interest. This is a mode to support functions not available on a real computer system. An example of this third mode is the function of interactively defining a new virtual machine configuration to be catalogued in a library of VM definitions.

The main technical issue to support these modes is that there needs to be a mechanism to easily shift among modes, such as typing special characters.

An additional special consideration for consoles is that messages from the operating system in the virtual machine will use the virtual device designators. The mapping between virtual and real could be done by special recognition processing for that operating system and run in the service

machine so that when messages appeared on the terminal, they would contain real device designators. Alternatively, the operator could use special commands to map between virtual and real designators.

The support of system consoles is a slow speed activity compared to other device I/O and therefore fits well into the service machine approach. Also, the third mode of use for system consoles fits well with the service machine concept since that mode needs several operating system services such as the memory capability of the file system for storing virtual machine definitions.

MISSION **of** **Rome Air Development Center**

will plan and conduct research, exploratory and advanced development programs in command, control, and communications (C³) activities, and in the C³ areas of information science and intelligence. The principal technical mission areas are communications, electromagnetic guidance and control, surveillance of ground and aerospace objects, intelligence data collection and handling, information system technology, atmospheric propagation, solid state sciences, microwave physics and electronic reliability, maintainability and supportability.
